UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior
Computer Science Department

Doctoral Thesis

A generic software architecture for portable applications in heterogeneous wireless sensor networks

Author: María Soledad Escolar Díaz
Supervisor: Jesús Carretero Pérez

March 2010

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior
Departamento de Informática

Tesis Doctoral

A generic software architecture for portable applications in heterogeneous wireless sensor networks

Autor: María Soledad Escolar Díaz
Director: Jesús Carretero Pérez

Marzo de 2010

TESIS DOCTORAL

A generic software architecture for portable applications in heterogeneous wireless wensor networks

Autor: María Soledad Escolar Díaz
Director: Jesús Carretero Pérez

Firma del Tribunal de Tesis:

Presidente:

Vocal:

Secretario:

CALIFICACIÓN:

Marzo de 2010

-¡Buenas noches! –dijo el principito.
-¡Buenas noches! –dijo la serpiente.
-¿Sobre qué planeta he caído? –preguntó el principito.
-Sobre la Tierra, en África –respondió la serpiente.
-¡Ah! ¿Y no hay nadie sobre la Tierra?
-Esto es el desierto. En los desiertos no hay nadie. La Tierra es muy grande –dijo la serpiente.
El principito se sentó en una piedra y elevó los ojos al cielo.
-Yo me pregunto –dijo– si las estrellas están encendidas para que cada cual pueda un día encontrar la suya. Mira mi planeta; está precisamente encima de nosotros... Pero... ¡qué lejos está!
-Es muy bella –dijo la serpiente–. ¿Y qué vienes tú a hacer aquí?
-Tengo problemas con una flor –dijo el principito.

Antoine de Saint-Exupéry,
El Principito.

Abstract

In the last years, wireless sensor networks (WSNs) are acquiring more importance as a promising technology based on tiny devices called sensor nodes or *motes* able to monitor a wide range of physical phenomenon through sensors. Numerous branches of science are being benefited. The intrinsic ubiquity of sensor nodes and the absence of network infrastructure make possible their deployment in hostile or, up to now, unknown environments which have been typically unaccessible for humans such as volcanos or glaciers, providing precise and up-to-date data.

As potential applications continue arising, both new technical and conceptual challenges appear. The severe hardware restrictions of sensor nodes in relation to computation, communication and specifically, energy, have posed new and exciting requirements. In particular, research is moving towards heterogeneous networks that will contain different devices running custom WSN operating systems. Operating systems specifically designed for sensor nodes are intended to efficiently manage the hardware resources and facilitate the programming. Nevertheless, they often lack the generality and the high-level abstractions expected at this abstraction layer. Consequently, they do not completely hide either the underlying platform or its execution model, making the applications programming close to operating system and thus reducing the portability.

This thesis focuses on the portability of applications in heterogeneous wireless sensor networks. To contribute to this important challenge the thesis proposes a generic software architecture based on sensor node, which supports the process of applications development by homogenizing and facilitating the access to different WSN operating systems. Specifically, the next main objectives have been established.

- Designing and implementing a generic sensor node-centric architecture distinguishing clearly the different abstraction levels in a sensor node. The architecture should be flexible enough in order to incorporate high-level abstractions which facilitate the programming.
- As part of the architecture, constructing an intermediate layer between applications and sensor node operating system. This layer is intended to abstract away the operating system by demultiplexing a set of homogeneous services and mapping them into operating system-specific requests. To achieve this, programming language extensions have to be also specified on top of the architecture, in order to write portable applications. In this way, platform-specific code can be generated from these high-level applications for different sensor node platforms. In this way, architecture deals with the problem of heterogeneity and portability.
- Evaluating the feasibility of incorporating the abstractions above mentioned within the development process in terms of portability, efficiency and productivity. In this environment the footprint is a specially critical issue, due to the hardware limitations. In fact, an excessive overhead of applications size could make prohibitive the proposed solution.

The thesis presents a generic software architecture for portable applications in heterogeneous wireless sensor networks. The proposed solution and its evaluation is described in this document. Theoretical and practical contributions of this thesis and the main future research directions are also presented.

Resumen

En los últimos años, las redes de sensores inalámbricas han adquirido cada vez mayor protagonismo y se han erigido como una prometedora tecnología basada en dispositivos pequeños denominados nodos sensores o *motes*, que son capaces de monitorizar fenómenos físicos a través de diferentes sensores. Un gran número de diferentes ramas de las ciencias podrían verse beneficiadas. La naturaleza ubicua de los nodos además de la ausencia de una infraestructura de red, hacen posible la instalación de estas redes en terrenos inhóspitos y típicamente inaccesibles para los seres humanos, como por ejemplo glaciares o volcanes, para proporcionar un conocimiento preciso y actualizado.

A medida que continúan apareciendo diferentes aplicaciones potenciales, surgen nuevos retos tanto técnicos como conceptuales. Las restricciones severas de los recursos en términos de cómputo, comunicación y, sobre todo, energía, plantean nuevos requerimientos. En particular, la investigación tiende a crear redes heterogéneas que incluyen diferentes dispositivos de hardware e integran sistemas operativos desarrollados *ad-hoc*. Los sistemas operativos específicamente diseñados para nodos sensores han sido concebidos para gestionar eficientemente sus recursos de hardware y facilitar la programación. Sin embargo, a menudo carecen de la generalidad y de las abstracciones de alto nivel esperadas en esta capa de abstracción. Por tanto, los sistemas operativos no enmascaran completamente su modelo de ejecución ni la plataforma subyacente, convirtiendo la programación de aplicaciones en fuertemente acoplada al sistema operativo y, consecuentemente, reduciendo la portabilidad.

Esta tesis se centra en la portabilidad de aplicaciones en redes de sensores inalámbricas heterogéneas. Con el objeto de contribuir a este relevante ámbito de estudio, la tesis propone una arquitectura de software genérica basada en nodo sensor, la cual soporta el proceso de desarrollo de aplicaciones homogeneizando y facilitando el acceso a diferentes sistemas operativos de nodos sensores. Específicamente, se han establecido los siguientes objetivos principales:

- Diseñar e implementar una arquitectura genérica de nodo sensor distinguiendo con claridad los diferentes niveles de abstracción del nodo sensor. La arquitectura propuesta debería ser flexible para poder incorporar nuevas abstracciones de alto nivel que faciliten la programación de las aplicaciones.

- Como parte de la arquitectura, deberá construirse una capa de abstracción localizada entre las aplicaciones y el sistema operativo. Su objetivo es abstraer el sistema operativo subyacente mediante un conjunto de servicios homogéneos que puedan ser mapeados en servicios específicos del sistema operativo. Para ello se deberá especificar en la capa superior de la arquitectura el conjunto de extensiones del lenguaje de programación que permitan escribir aplicaciones portables. Consecuentemente, el código específico de la plataforma puede ser generado a partir de las aplicaciones de alto nivel para diferentes plataformas de nodos sensores. De esta manera, la arquitectura trata los problemas de portabilidad y heterogeneidad en la construcción de aplicaciones.

- Evaluar la factibilidad de incorporar las abstracciones previamente mencionadas para ser usadas dentro del proceso de desarrollo de aplicaciones, en términos de portabilidad, eficiencia y productividad. En el entorno de las redes de sensores, el consumo eficiente de los recursos de hardware es un aspecto crítico debido al presupuesto limitado del hardware. De hecho, una sobrecarga excesiva haría prohibitiva e inviable la propuesta.

Esta tesis describe una arquitectura de software genérica para aplicaciones portables en redes de sensores inalámbricas heterogéneas. La solución propuesta y su evaluación se presentan en este documento. Las contribuciones teóricas y prácticas de esta tesis serán analizadas, así como las líneas futuras de investigación que derivan de este trabajo.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Symbols and terminology

A	Set of Architectures
AM	Active Message
APP_{SN_OSAL}	Set of applications written using SN-OSAL
BNF	Backus-Naur Form
C	Components
Contiki	The Operating System for Embedded Smart Objects
E	Events
H	Set of Hardware platforms (in the architecture proposed)
HW	Hardware
I	Interfaces
MDA	Model Driven Architecture
O	Set of operating systems (in the architecture proposed)
OS	Operating System
OSAL	Operating System Abstraction Layer (in the architecture proposed)
$OI_{Contiki}$	Contiki interface
OI_{T1}	TinyOS 1.x interface
OI_{T2}	TinyOS 2.x interface
Portability	Independence degree that applications present with respect to the execution platform
Productivity	Efficiency indicator relating the amount of product obtained and the resources involved
Protothreads	A blocking event handler in the Contiki operating system
Rime	Sensor network protocols implemented in Contiki
S	Services
SENFIS	SENsor node File System
SN-DSL	Sensor Node Domain Specific Language
SN-OSAL	Sensor Node Open Services Abstraction Layer
$SN\text{-}OSALI$	SN-OSAL interface
TinyOS	Tiny Microthreading Operating System
T1	TinyOS 1.x
T2	TinyOS 2.x
$\mathcal{TP}$	Transformation Process
uIP	Lightweight IPv4 version for sensor nodes implemented in Contiki
Usability	Capability of a system to be used
VisualOSAL	Graphical development framework for building SN-OSAL applications
W	Wirings
WSN	Wireless Sensor Network
ZigBee	Communication standard for WPANs
ESB, SKY, Mica, Telos	Some sensor nodes
$\diagdown_P$	Portability function
$\diagdown_T$	Translation function
$\mathcal{R}_P$	Portability relation
$\mathcal{R}_T$	Translation relation

Chapter 1

Introduction

This chapter introduces the context of the work presented in this thesis. The first section provides an overview on wireless sensor networks. The second section presents a brief description of the motivation and of the wireless sensor networks challenges that need to be addressed within the scope of our work. The next section discusses the specific goals of the thesis work. We conclude the chapter providing a road map for the rest of this document.

1.1. Thesis definition and scope

The past decades have witnessed a powerful digital revolution which has drastically affected the society at many levels including our everyday life. From Internet to the most modern mobile phones, the hardware basic component is the microprocessor. The 2% of the microprocessors currently sold are used in personal computers, while the 98% remaining are used by embedded systems [Dun07b]. These embedded microprocessors impose special restrictions such as scarce memory and compute capacities.

In recent years, wireless sensor networks (WSNs) have emerged as a promising technology in the field of embedded systems. In 2003 –when the WSNs were only an incipient research field– the prestigious journal *The MIT Technology Review* published an article envisioning that this technology would change the world [CHA03]. Only a few years later, the growth experienced by WSNs began to be compared to the Internet revolution [Pin04]. Many research institutions around the world have focused their efforts in topics related to WSN technology.

Despite the great expectation and the clear potential that WSNs could achieve, experts assure that we are in fact very far from being able to make massive use of this technology. Meanwhile the number of potential application fields, as well as the number of applications within these fields continues to increase to include the natural environments, industry, health, civil engineering or security.

A wireless sensor network connects the physical and computational world by monitoring a wide variety of environmental phenomena through devices called sensor nodes or *motes*. Most of the environments for which WSNs have mainly been conceived (e.g. figuring out a WSN for climatic change monitoring deployed from an aircraft) are inhospitable or unaccessible; and the replacement of the batteries for these devices is not be possible or at least, the manual process is extremely complex. Therefore, it is crucial to ensure that devices do not consume large quantities of energy. For this reason minimizing energy consumption has become the highest priority within the WSN research community, with the study of efficient-energy network algorithms being one of

the topics most in demand.

Figure 1.1 shows a MicaZ sensor node. Its tiny size greatly restricts the physical resources and, consequently, determines their physical capabilities.

Figure 1.1: MicaZ sensor node or *mote*. Dimensions are around 58 x 32 x 7 millimeters excluding batteries.

However, there are any other open challenges. Due to the specific nature of the devices, and the particular conditions of the environments where they are deployed, WSNs continue posing new and exciting challenges such as system integration and operating systems. To deal with the heterogeneity, the most general trend has been the design of middlewares, which support the development process at the network level. However, this solution has usually addressed different requirements in a particular scenario, but lacks the completeness supposed at this level of abstraction.

Focusing on the WSN operating systems, different samples specifically designed for sensor nodes have been proposed in recent years to deal with the particular features of sensor nodes. TinyOS, developed at the laboratories of the University of Berkeley by Phillip Levis under the supervision of David Culler in 2002, is *de facto* standard operating system. Another relevant approach is Contiki, developed in Europe at the Swedish Institute of Computer Science (SICS) by Adam Dunkels, as the leader of the project, in the year 2003. Among the main functions of WSN operating systems, the efficient management of the physical resources and the supply of high-level abstractions to simplify the programming should be emphasized. Regarding to the second point, due to the hardware heterogeneity and the diversity of the different application domains, operating systems could be forced to make wide interfaces available as well as distinct implementations which are more suitable for each specific problem. Given the limited hardware capabilities of sensor nodes, it is not always feasible.

Consequently, heterogeneity and the small size of sensor nodes convert applications development into a hardware-coupled task. Developers must be aware of both functional and non-functional requirements. Usually, they have to perform implementations at different layers of the software architecture: from the hardware to the high-level application itself, including driver programming or network protocols. Integration is also complex. For these reasons, the software developed for sensor nodes is typically *ad-hoc* and, subsequently, difficult to reuse and integrate into other systems. It should also be robust and reliable, and frequently must incorporate features related to network adaptability, reprogrammability and security.

1.2. Motivation

Wireless sensor network research is moving towards heterogeneous networks that will contain different devices running custom operating systems. Developer teams work in parallel using both proprietary and commercial sensor nodes, which hold several physical devices. Hardware in sensor nodes are becoming more and more versatile, influenced by new requirements such as energy saving or sensors that would make possible to develop new applications.

Heterogeneity has been traditionally masked by the operating system, which hides the low-level details of the devices and their management. Due to the severe hardware restrictions, operating systems do not always integrate the most efficient mechanisms for the whole set of platforms and application domains. Specifically, operating systems designed for sensor nodes have not been conceived to address some relevant challenges:

- The abstraction level of applications with respect to the underlying levels is very low because the programmers must explicitly call the hardware and software components required by the applications. They must therefore have an exhaustive knowledge of both the hardware and operating systems.
- There is no clear division in architectural layers with different abstraction levels and, subsequently, there are no well-established roles and responsibilities for each layer. Furthermore, the boundary between hardware and software is ambiguous and still an active research area [CDE+05, LC02].

As shown in previous section, the operating system is not able to provide enough high-level abstractions to programmers, who are forced to perform tasks at different architectural levels: hardware, network protocols, or even the operating system, besides the applications itself. The restricted capabilities of the sensor nodes prevent the generality and abstractions that should be expected at the operating system level. Additionally, the specific operating systems execution models contribute to increasing heterogeneity and complexity, because applications programming is typically accomplished very close to the operating system. For these reasons, development is usually and *ad-hoc*, tricky and error-prone task and can be pointed out as one the factors currently reducing the set of people able to use the technology.

Facilitating the applications writing is another task concerning operating systems. In this sense, there are no programming languages defined on WSN operating systems intended to facilitate the node-centric programming, and, subsequently, the WSN applications are developed using the same programming language that the underlying operating system uses.

Moreover, portability becomes another critical problem enormously hindering the work of developers which incurs in long development times and effort. Focusing solely on the application level, one program might need to be ported to execute on a platform different than those for which it was initially conceived.

In addition to the previous factors, the lack of a standard API, and of generic architectures that encapsulate the difficulty exposed by the lower levels should be mentioned.

Subsequently, software developed for WSNs is frequently labeled as monolithic, *ad-hoc*, and platform-dependent. The consequences derived from this problematic clearly impact on the development time, increase the resources involved, and delaying the evolution of WSN technology.

In summary, designing, implementing, and maintaining software for sensor nodes is definitely not a trivial task [Lev06] [NAD+02] [MPar] [SG08] and, subsequently, more contributions should be made in this area. This thesis is based on this assumption.

1.3. Objectives

To approach the above mentioned challenges, an architecture dealing with the deficiencies previously presented is necessary. This thesis addresses these challenges and establishes as its main objective *the design, implementation and evaluation of a platform-independent architecture for writing generic and portable WSN applications in heterogeneous environments, which can be easily transported among different platforms*.

The thesis will study the software architecture of a *generic* sensor node. In particular, it will analyze the physical devices to extract their settings, functionality, and interface. This thesis will go into detail on several operating systems specifically designed for sensor nodes, which efficiently manage their hardware resources. Taking as a reference the previous prerequisites, this thesis proposal is aimed at exploring the feasibility of composing a flexible sensor node architecture, in which additional features can be incorporated to deal with the heterogeneity and applications portability.

More specifically, we are able to enumerate the following objectives for this thesis:

- Design and implementation of a multi-layered software and sensor node-centric architecture clearly distinguishing the different abstraction levels in a sensor node:
  - At the hardware level, a reasonable set of physical devices have to be studied in order to elaborate a generic and flexible hardware model, including the analysis of the resources and their properties, functionalities, and services.
  - At the operating system level, several operating systems for sensor nodes will be considered and analyzed: TinyOS (versions 1.x and 2.x) and Contiki. In particular, the primitives providing different functionalities will be identified because they will represent the connectors between each two architectural layers. As in the previous case, a generic model describing this level will be stated.
  - An abstraction layer for heterogeneous WSNs operating systems will be proposed and implemented. This intermediate layer is intended to homogenize the access to the underlying architecture through a standardized interface which masks the heterogeneity and complexity. This layer will act as an integrator element located between the application and the operating system level, encapsulating the OS services into bigger grained operations. To achieve this, a set of homogeneous services for sensor nodes will be proposed. The translation between these services and the platform-specific services will be performed automatically at this level, and in a transparent way for developers. Subsequently, equivalent source code is generated for a reasonable set of sensor node platforms.
  - At the highest level, the programming of platform-independent applications on top of the architecture proposed must also be addressed. The programming language extensions necessary to develop portable applications on top of the previous layer have to be described. To achieve this, a Domain-Specific Language has been created and its syntax and semantics must be analyzed.
- Evaluation of the proposed architecture in terms of portability, resource consumption, and productivity. For the first point, the degree of portability achieved will be analyzed taking into account the set of operating systems, sensor nodes and potential applications to be described using the architecture. In WSN environment, where the severe hardware constraints

constitute a critical factor for programming and deploying, quantifying the footprint and executable size of applications is a mandatory task. A set of demonstrators will be created to carry out the evaluation. Finally, the productivity estimation of different applications using the system proposed and other approaches will be computed and compared.

1.4. Structure of this document

This document describes the work developed in this thesis. This has been organized into nine chapters, whose contents are summarized in the following paragraphs:

- Chapter 1 has shown a brief description of the wireless sensor networks area, and it has presented the motivation for this thesis. The proposal of a generic software architecture for developing portable applications in WSN heterogeneous environments was stated as the main objective.

- Chapter 2, *State of the art*, analyzes the relevant aspects of WSN technology. In particular, hardware and operating systems specifically designed for sensor nodes are detailed, analyzing their execution models. The wide range of applications where the WSNs could help are briefly described, providing several examples. Wireless communication protocols and 802.15.4 IEEE standard are described. Background in software architectures for sensor nodes is also presented. Due to its importance for the development of this thesis, related work in programming approaches is extensively analyzed and different samples of each one are described.

- Chapter 3, *Problem statement*, clearly establishes the motivation for this thesis, to later be able to formulate the hypothesis of work, which will guide the development of this document.

- Chapter 4, *Sensor node-centric architecture*, presents the fundamentals of the architecture proposed. Firstly, its design is carry out in order to describe the basic concepts, components and integration mechanisms among them. Secondly, the mathematical formalization of the architecture is stated. The architectural layers are extensively studied, instantiating them with specific components which can be easily incorporated following the basis described.

- Chapter 5, *Operating System Abstraction Layer*, describes the greatest contribution of this thesis, consisting of an abstraction layer located on top of the WSN operating systems in order to homogenize the access the underlying architecture. *Sensor Node Open Services Abstraction Layer* (SN-OSAL) has been conceived to be a specific instance of this component. SN-OSAL is a translation layer specified in accordance with the previously established description, and intended to abstract away the underlying architecture thus facilitating the WSN applications building. In this way, SN-OSAL is able to generate platform-specific code in a way that is transparent for programmers.

- Chapter 6, *Application layer*, explains the syntax and sematics for applications development using the underlying SN-OSAL. To achieve this, a Domain Specific Language (DSL) has been specified for programming portable applications on top of SN-OSAL: the *Sensor Node Domain Specific Language* (SN-DSL). The syntax of the SN-DSL will also be stated through the definition of a BNF grammar able to recognize it unambiguously.

- Chapter 7, *Graphical development framework: VisualOSAL* presents a tool for graphical development denominated *VisualOSAL*, which allows the visual composition of applications, and additionally addresses the life cycle for WSN applications building developed on top of the architecture proposed.
- Chapter 8, *Evaluation*, presents the experiments performed to measure the portability degree, footprint and productivity of the architecture proposed. Results obtained to provide an idea about the efficiency of SN-OSAL and its feasibility to be incorporated into the traditional WSN architecture.
- Finally Chapter 9, *Conclusions*, shows the main theoretical and practical contributions derived from the elaboration of this thesis. The chapter also describes the future work and main research directions, which would take this work as basis.

Three appendixes finish this document. Appendix A offers examples of hardware components characterization through the definition of XML Manifests and Schemas described in Chapter 4. Appendix B lists the mapping rules used by SN-OSAL (described in Chapter 5) in order to carry out the translation between its interface and each OS-specific interface. Appendix C includes the description of SN-OSAL API, which is composed of the services designed to abstract away the equivalent ones into the different WSN operating systems.

Finally, the bibliography used for the elaboration of this thesis is included, in addition to several web pages for recommended reference.

Chapter 2

State of the art

This chapter covers the basis of wireless sensor networks and the background necessary to help the reader understand the remaining chapters. Due to the fact that wireless sensor networks are themselves a relatively new technology posing challenges which have not been described up to now, general concepts and definitions are stated. An overview of a typical device architecture is given, splitting it into hardware and software pieces which are extensively described. Since communication plays a critical role, reference work in this area is gathered. In addition, tools assisting several stages of the development process are identified. The Model Driven Architecture (MDA) standard is introduced, as a basis of work presented later. Finally, a summary of the most relevant aspects described ends this chapter.

2.1. Wireless sensor networks

A wireless sensor network (WSN) is an *ad-hoc* network consisting of cooperating, autonomous, small-sized devices termed sensor nodes or *motes* usually connected wirelessly amongst themselves to achieve a communication *gateway* with the capacity to forward data through any other technology, or a *base station*. The potential of these devices is based on the integration of different sensors, from simple sensors (e.g. temperature, light, humidity) to complex sensors (e.g. GPS, imagers, microradars), which make it possible to measure a wide range of environments and to monitorize physical phenomenon, providing users with accurate and up-to-date knowledge.

The tiny size of sensor nodes facilitates their installation and management, but also determines their physical and logical capabilities. Sensor nodes are severely resource constrained in terms of communication, computation, storing capacity and most of all, energy. Due to the specific nature of the devices, and the special conditions of the environments in which they are deployed, the WSNs continue posing challenges for the scientific community.

Usually sensor nodes integrate a low-power microcontroller including RAM (for data) and ROM (for code) memory chips of small capacity and one analogical-digital converter (ADC). A radio transceiver allows sending and receiving data to or from other similar devices. Different kinds of sensors can be attached directly to the device or *sensor board* which is connected to the mote through an expansion connector. The energy source typically consists of two conventional batteries. The lifetime of the network depends on the lifetime of each one of its nodes, and therefore, good use of the hardware components is always required, besides correct programming, which results in maximization of the life of the batteries.

Sensor nodes enable a small application to be loaded inside the microcontroller. Usually the

application code includes the complete embedded operating system and the user program, which performs several tasks as a response to certain events. The typical behavior of a program consists of the following: to save resources, most of the time the sensor node is in the *sleep* state. When an event happens (e.g. a timer, external data), it wakes up and goes to the *active* state. In this state, mote could perform a set of simple actions such as monitoring an environmental phenomenon through sensors, or forwarding data into the network via the radio. After that, nodes are put in a lower energy mode (*sleep* state) until the cycle starts again.

The benefit of the network lies in the distributed task instead of the isolated nodes operation. Usually nodes are continuously exposed to extremely unfavorable conditions and changing environments, which means they must self-configure and operate in an unattended way. Nodes are distributed over the network in an *ad-hoc* way. That means that there is no statically defined network structure or topology and any node must support a multi-hop routing algorithm which allows data to be forwarded.

2.1.1. Definitions

- A *sensor network* is a distributed computing system where some or all nodes are capable of interacting with the physical environment [BP08].
- A *wireless sensor network* (WSN) is a sensor network where communication among nodes is wirelessly performed.
- A *sensor node*, also known as a *mote*, is a node in a sensor network capable of performing some processing, gathering sensory information and communicating with other connected nodes in the network.
- A *WSN operating system* is an operating system specifically designed for sensor nodes, taking into account the special characteristics of the devices such as low-power consumption or low-rate data.
- An *ad-hoc network* is a local area network (LAN) that is built spontaneously as devices connect. Instead of relying on a base station to coordinate the flow of messages, the individual network nodes forward packets to and from each other.
- An *heterogeneous WSN* (H-WSN) is a sensor network where more than one type of node is integrated.

2.1.2. Emerging applications

There are many extremely different scenarios where one sensor network can collect precise and long-term data, increasing the knowledge that scientists have about the physical world. Typical WSN applications are also multi-disciplinary, since data must be analyzed by experts in geology, biology and other branches of science. Below, applications are grouped taking into account their application domain. Additionally, some examples of the benefits of this technology are given.

- In military environments the first scenarios of application were found. Modern research may have started around 1980 with the *Distributed Sensor Networks* (DSN) project of the Defense Advanced Research Projects Agency (DARPA) [CK03]. There are probably many similar projects focused on this research subject. Currently, for this field the systems for

intrusion detection can be mentioned, whose objective is movement detection close to frontiers where it is installed.

- The environmental field represents the biggest exponent of WSN applications. A great diversity of applications can be found in this area from wildlife tracking to geological process monitoring. The *Great Duck Island* project [MCP+02] deployed a sensor network composed of 32 nodes on a small island of the coast of Maine (USA) to study the habitat and quantify the impact of human presence on the animals and plants. The result of this research will allow the creation of a habitat monitoring kit, which can be used by scientists and researchers in other fields. The *ZebraNet* project [JOW+02] tracks the behavior of zebras in Kenya, using customized tracking collars. These kinds of applications have specific conditions such as long distance monitoring and the mobility of nodes. The improvement of the agricultural process is another application of interest. Vineyard monitoring [BBB04, Gal06] is probably one of the pioneer WSN applications which has reached maturity. Fire detection and studies about volcanos [WALJ+06] are also other examples of applications in this field.

- The civil and structural engineering field offers many application scenarios for WSNs. Civil structures have a predetermined lifetime, but however due to causes such as building defects or internal building environment, they suffer a rapid and progressive debilitation. These kinds of applications are related to the design, construction and life-long maintenance of civil engineering structures. Sensors can take measurements as to the deterioration level of concrete, steel, masonry, and composite materials. A relevant example is the *Sustainable Bridges* project [CHAh], whose goal is to study damages, and inspect and diagnose the level of deterioration of the European railway bridges.

- The health and medicine field is acquiring more importance because of sensors allowing measurements of vital signs such as blood pressure, pulse and respiration rate or body temperature. Patients of different pathologies can be monitored at home, through a WSN in real time, sending samples to a central computer where they are analyzed by experts which can quickly react in emergency cases. *CodeBlue* [MFjWM04] is a project developed at Harvard University for medical care using this technology. The goal of the project is to explore different medical applications including pre-hospital and in-hospital emergency care, disaster response, and patient rehabilitation.

- Industry is other wide area of application, ranging from quality control to transport and logistics management process [EBMP+05]. It is possible to find projects related to quality management, machinery diagnosis or variable monitoring which seek to reduce wiring and costs.

In [RM04] a set of real WSN projects located in Europe are exhaustively examined, with the final goal of defining the design space of WSNs and elaborating some classification criteria attending to parameters such as the topology, cost, heterogeneity, and mobility of their nodes.

2.1.3. Challenges, trends, and limitations

The popularity acquired by the WSNs in the worldwide research community has increased dramatically in the last years, comparable only to the Internet revolution [Pin04]. Although important advances have been achieved, maturity is still far away, and unclear, and there are still many open questions in multidisciplinary areas: communication, computation, and hardware.

In relation to miniaturization, microelectronic research is intended to explore a new generation of tinier and cheaper sensors. The term *Smartdust* is used to describe a network composed of *Micro Electro Mechanical Systems* (MEMS) [KKP99] with a capacity of wireless communication. MEMS devices are associated to micromachines composed of different components such as a unit of processing, wireless communication, and several microsensors integrated on a common silicon substrate. These devices are intended to be extremely small, ranging in size from a micrometer scale to a millimeter (it can be compared to a grain of sand or even a dust particle). Miniaturization plays an important role because in that size scale, the properties of materials and the concepts of classic physics are not always true. MEMS are also intended to minimize the power consumption of the current sensor nodes. Figure 2.1 shows the Emkay SiSonic microphone [Aco02], which has been produced using the MEMS technology.

Figure 2.1: MEMS devices: SiSonic microphone developed by Emkay Innovative Products [Aco02].

Low-energy operation modes is a key factor in extending the lifetime of the network. In this area, research is focused on not only energy-efficient networking solutions (e.g. network protocols, aggregated data, in-node processing), but on reducing the number of duty cycles spent by microcontrollers (that is, maximization of time in which the mote is in the *sleep* state). Alternative power sources have stirred a great interest as energy suppliers, such as solar cells or harvesting technology, which allow recharging the small size batteries of sensor nodes.

The dynamic nature of WSNs (e.g. network conditions, neighborhood, location, size) and sensor nodes (e.g. constrained, autonomous, adaptive, unattended, in many cases inaccessible), in addition to the inhospitable terrains where they are deployed, have brought into question the advisability of paradigms traditionally accepted for networked systems. In this sense, protocols and algorithms have been redesigned to meet the new requirements.

The feasibility of achieving generic software architectures has been questioned due to the hardware heterogeneity, the limitation of resources, and the application domains. Moreover, it is still an unanswered question as to if it would be feasible to have a common wireless sensor network architecture, such as has been presented in [HKKW03], [CDE+05].

Software development is frequently *ad-hoc*, and performed from a bottom-up perspective. which means that, in some cases applications portability has been sacrificed. Although the sensor node applications are supported by an operating system which manages the underlying hardware, while hiding its complexity, they do not always provide high-level abstractions for making the programming easy and portable.

Ubiquitous computing [WB97], "*the calm technology that recedes into the background of our lives*" in the words of Mark Weiser, considered the father of the technology, find a faithful ally in the wireless sensor networks. According to the Weiser's ideas, "*the best computer is a quiet, invisible servant*" and "*the computer should extend your unconscious*". Sensor nodes match that description very precisely. However, the potential for wireless sensor networks and ubiquitous computing is greater than what has been explored up to now [LL05].

2.2. Wireless sensor network architecture

A wireless sensor network integrates several hardware entities. Figure 2.2 shows the architecture of a typical WSN, where the following elements can be clearly distinguished: *mote* (M), *sensor board* (S), *sensor node* (M+S), and *gateway* (G).

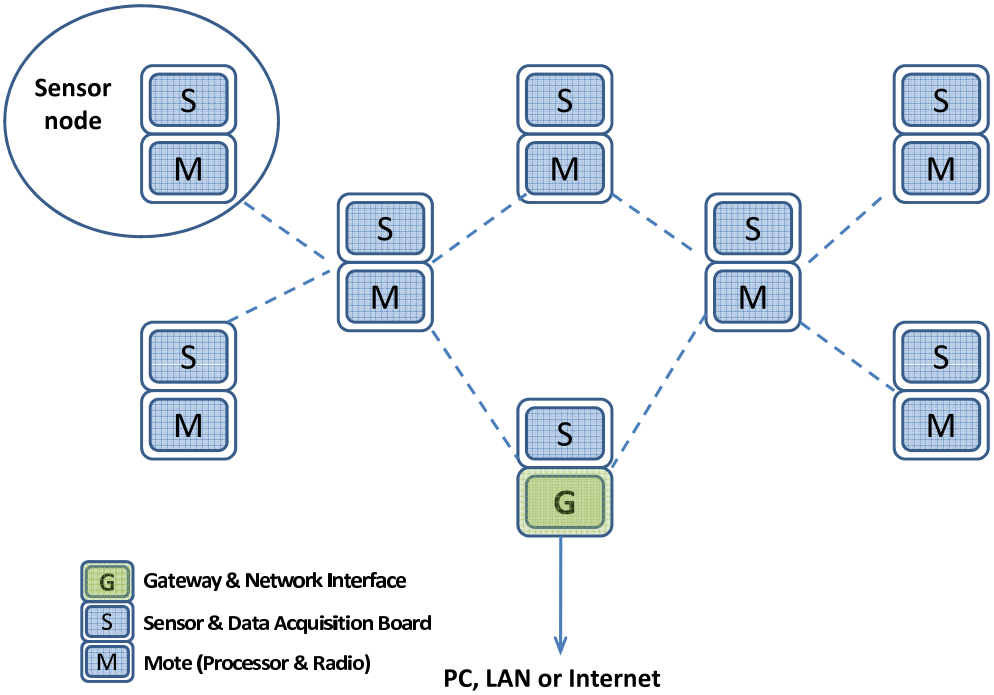

Figure 2.2: Wireless sensor network architecture.

Next, a more rigorous definition is given:

- *Sensor node* is composed of a mote and a sensor board. *Mote* is the entity composed of a processor and radio devices. *Sensor Board* is a data acquisition board attached to the mote through an expansion connector, which includes a set of sensors. Some sensor node models incorporate the sensors into the mote itself (e.g. Telos family motes). Some examples of sensors are temperature, humidity, accelerometer, magnetometer, or sound.
- The *gateway* has a dual function:
  - *Programming board* allows downloading the binary code from the PC to the mote microcontroller, typically connected through a serial interface.
  - The *gateway* provides the interface between the wireless network and the external world. Typical gateways incorporate a RS-232 or USB port (e.g. MIB510 and MIB520 gateways respectively), which is used to forward data to the PC, or an Ethernet port (e.g. MIB600 gateway) to forward data to the Internet.
- Optionally, a WSN can use *Stargate* nodes, with a function similar to Ethernet gateways. They may integrate a memory and processor of larger capacities typically running a Linux operating system, for temporal storing or data processing.

- The *base station* is the embedded or conventional device intended to collect data proceeding from the WSN for further data processing or analysis.

Figure 2.3 shows examples of components in a WSN: Telos model mote, the MDA300CA sensor board including temperature and humidity sensors, and the MIB520 gateway, with USB interface to connect to the PC. In the next section, a more detailed explanation about the hardware components integrated into sensor nodes is given.

Figure 2.3: Hardware components manufactured by Crossbow Technology, Inc. From left to right: Telos mote, MDA300CA sensor board and MIB520 gateway.

2.2.1. Hardware technology

Hardware is currently an active research area carried out in universities around the world and in private companies. The possibilities in this field are enormous because of the increasing need to look for new sensors for different applications, the advances in miniaturization, components to be integrated (e.g. GPS), or new features to save energy.

At the University of Berkeley earlier modern WSN research took place. Since the first mote saw the light in 1998 at this university, several companies have been manufacturing and commercializing hardware and software for WSNs. Up to now, Crossbow Technology, Inc. [CHAd] has been the major supplier of wireless sensor technology with more than 4000 customers worldwide. It was founded in 1995, but it was not until 2001 when the company became the first commercial provider of original *motes* of Berkeley. Besides hardware, Crossbow provides complete development kits and 3-tier software solutions: mote, client and server. Although it is far from achieving the business volume of Crossbow, another recently created company must be mentioned: Sentilla [CHAg], founded in 2007 by researchers of University of Berkeley. Sentilla has developed a Java architecture in different layers and produces hardware and software on demand for WSN applications.

As mentioned in the previous section, a sensor node is a device with computation, communication and sensing capacities. Typically, it is composed of a mote and a sensor board. However, some platforms incorporate a set of sensors inside the mote board itself, such as Telos family motes (see Telos mote in Figure 2.3), and therefore, a sensor board is not present. In other cases, when a sensor board is required, such as by Mica family motes, the connection between the sensor board and the mote is usually done through an expansion connector. Figure 2.4 shows the 51-pin expansion connector for IRIS and Mica family motes, except for Mica2Dot mote, which uses a 19-pin circular expansion connector.

The expansion connector provides an interface for both sensor boards and gateways. The connector includes interfaces for power and ground, power control of peripheral sensors, ADC inputs for reading sensor outputs, UART and I2C interfaces, general-purpose digital I/O, and others.

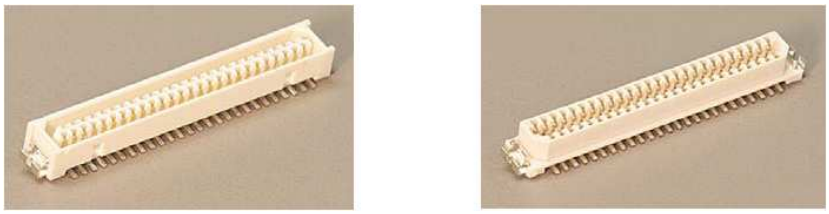

Figure 2.4: The 51-pin expansion connector. It represents the communication interface between the sensor board and the mote, and also between the mote and the gateway for system programming [CHAd].

2.2.1.1. Motes

A typical *mote* presents the generic hardware architecture represented in Figure 2.5. As shown, it is composed of a set of hardware components which are described as follows:

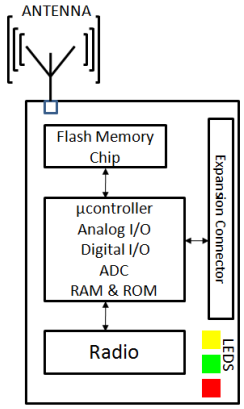

Figure 2.5: Hardware architecture of a mote.

- *Microcontroller* represents the low-capacity processor (e.g. Atmel Atmega 128L [Atm08], MSP430 [Ins08]) which usually operates at low frequencies (e.g. 7 MHz) and has an architecture from 4-bit to 32-bit. Also, it contains different RAM and ROM memories, an Analogical-Digital Converter and different *clocks* in order to allow local timing. Some examples of microcontrollers are:

  - Atmega128L [Atm08] from ATMEL, a low-power CMOS 8-bit microcontroller based on the RISC architecture. This microcontroller has 32 general purpose working registers. The memory system is composed of 128 KB of In-System Programmable Flash, 4 KB EEPROM, and 4 KB SRAM. The system clock is composed of several clocks allowing four timers: two 8-bit and two 16-bit width timers. The micro also incorporates a 10-bit ADC. The microcontroller supports six software selectable energy saving modes. This microcontroller is used in Mica2 and MicaZ motes, among others.
  - MSP430 [Ins08] from Texas Instruments, an ultralow power 16-bit RISC CPU with 16-bit registers achieving speeds up to 4 MHz. It holds a memory system composed of

2 KB RAM memory chip, and a 60 KB ROM memory chip, both containing code and data. The clock system is supported by the basic clock module that includes support for a 32768 Hz watch crystal oscillator, an internal digitally-controlled oscillator (DCO) and a high frequency crystal oscillator. The modules are capable of generating 16-bit timers. The ADC12 is the module responsible for performing 12-bit analog-to-digital conversions at the very high rate of 200 thousand samples per second. The MSP430 microcontroller has one active mode and five software selectable low-power modes of operation. This microcontroller is included in Telos and Eyes motes, among others.

- The *radio* device (e.g. CC2420 radio from Chipcon Products) provides wireless communication to the sensor node, and supports the WSN specific communication properties such as low energy and data rate, and short distances. Some radio devices for motes are the following:

  - TR1000 [CHAi] transceiver is manufactured by RF Monolithics, and it was the first radio device integrated into a sensor node (weC). In fact, it was present until the first generation of Mica motes. The main feature is its low consumption both for transmitting and receiving, especially in the sleep mode. TR1000 uses two frequency bands: 868 and 916 MHz.
  - CC1000 [CHAb] radio from Chipcon operates in a frequency selectable from 300-1000 MHz, but is mainly intended for 315, 433, 868 and 915 MHz frequency bands. Low consumption is one of the most attractive characteristics of this device. It is held in Mica2 and Mica2dot motes.
  - CC2400 [CHAc] radio device is one of the first radio Zigbee-compliant integrated into Crossbow devices. This radio device achieves a data rate of 250 Kbps (which means 6.5 times more than its predecessor), but as a disadvantage, its consumption is notably higher. This device is held in MicaZ and Telos family motes.
  - nRF2401 [CHAf] radio transceiver from Nordic Semiconductors allows transmitting and receiving at the 2.4-2.5 GHz ISM band, and therefore is ZigBee compliant. One of its advantages is that it can achieve the highest data rate to date (1 Mbps), in addition to its low power consumption. This device is hosted into Sensor Cube [DRD+07] platforms.

- *Memory* is an external device with longer capacity than the internal memories, capable of storing temporal data provided by different sources (sensors, network or logs). In the following, the most popular flash memory chips integrated into sensor nodes are described:

  - Atmel AT45DB [CHAa] is integrated into Mica family and TelosA motes, and it has a total capacity of 512 KB. Flash memory chips are usually divided into sectors. In this case, the flash is divided into 128 KB sectors. Every sector is itself divided into pages, and each page is 264 bytes long (256 bytes for data, 8 bytes for meta-data).
  - ST M25P40 [CHAe] flash memory chip is included into TelosB and Eyes platforms. This device presents similar settings to the previous one: It has a capacity of 4 Mbit, and it is organized into 8 sectors, each one containing 256 pages that are 256 bytes long.
  - Intel Strataflash is hosted in Intel Mote2. It is the lowest cost-per-bit NOR memory chip, with a capacity of 32 megabytes, which are divided into 128 KB sectors.

- *Battery* provides energy to the sensor node (e.g. alkaline batteries). Motes usually hold two conventional batteries as power supplier. Numerous research projects focus on alternatives for energy harvesting, which are typically based on solar cells.
- *LEDs* (Light-Emitting Diode) are attached to the mote board with the main purpose of debugging. Typically, there are three LEDs integrated into a sensor node (red, green and yellow) although in some motes, an additional blue LED has been added.
- A *sensor board* usually contains several sensors and actuators which are able to sense the physical environment. A more detailed explanation is given in the next section. When the sensor board is present, the *expansion connector* acts as a bridge between the sensor board and the mote microcontroller. As mentioned, the most typical connector is shown in Figure 2.4.
- *I/O buses* transport internal data between physical components (microcontroller, radio and memory) in accordance with a specific I/O protocol. Different interfaces coexist in a sensor node (e.g. Serial Peripheral Interface (SPI), Inter Integrated Circuit (I^2C) and Universal Asynchronous Receiver/Transmitter (UART)).

Since WeC, the first mote manufactured, Crossbow has launched into the market more than a dozen of motes (and related hardware, such as sensor boards and gateways), with different components and features. Tables 2.3 and 2.4 describe the sensor nodes evolution in a timeline in detail.

2.2.1.2. Sensor boards

The sensor board is a device present in the Mica family motes. Sensor boards integrate several sensors and actuators which are able to sense different phenomena from the physical environment. Raw individual measurements can be obtained on demand from each sensor, and forwarded to the microcontroller. Table 2.1 identifies some sensor boards commercialized by Crossbow, the specific sensors integrated on them, compatible motes, and the price in dollars.

SENSOR BOARD	Sensors	Motes	Price ($)
MTS300	Light,Temperature,Acoustic,Sounder	Mica2,MicaZ,Iris	128
MTS310	Light,Temperature,Acoustic,Sounder,Dual-Axis Accelerometer and Magnetometer	Mica2,MicaZ,Iris	250
MTS400	Light,Temperature,Humidity,Barometric,Pressure Seismic	Mica2,MicaZ,Iris	284
MTS420	Light,Temperature,Humidity,Barometric,Pressure Seismic,GPS	Mica2,MicaZ,Iris	404
MDA100	Light,Temperature,Prototype area sensor	Mica2,MicaZ,Iris	101
MDA300	Temperature,Humidity	Mica2,MicaZ	236
MDA320	Up to 8 Channels of 16-bit Analog Input	Mica2,MicaZ	-[1]

Table 2.1: Sensor boards settings.

2.2.1.3. Gateways and stargates

Gateways and stargates are devices providing the capacity of communication between the WSN and the external world. This feature implies a dual functionality: on one hand, the program-

[1] Discontinued product.

ming of motes consists of downloading the binary code from the PC to the mote microcontroller; and on the other hand, the routing of the data obtained inside the WSN to the PC. For these issues both gateways and stargates could be used. Downloading the executable code into the microcontroller implies connecting the gateway to the PC through the proper port (e.g. USB) and attaching the mote to the gateway through the expansion connector. The application program must first be compiled in the PC and subsequently, it can be copied to the microcontroller memory. The transport of data captured inside the WSN and directed to the base station is carried out by both gateways and stargates. Stargates exceed the capabilities of gateways providing longer capacity of storage (32 MB of flash memory and 64 MB of SDRAM) and processing (a 32-bit, 400 MHz Intel PXA255 XScale RISC processor), offering the possibility of executing a traditional operating system such as Linux. Figure 2.6 shows the stargate produced by Crossbow Technology Inc. Table 2.2 presents the most commonly used gateways manufactured by Crossbow. As shown, different connectors are integrated to provide the dual functionality described.

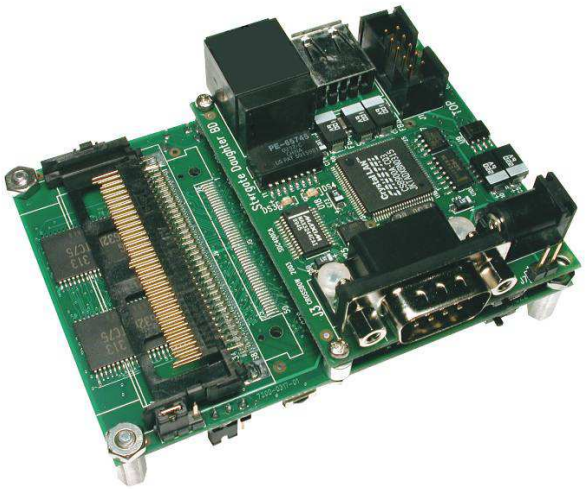

Figure 2.6: Stargate development platform: Processor board and daughter card.

GATEWAY	mib510	mib520	mib600
Compatible motes	IRIS	IRIS	IRIS
	Mica family motes	Mica2,MicaZ	Mica2, MicaZ
Programming interface	51-pin connector	51-pin connector	51-pin connector
	19-pin connector		
PC communication	RS-232 serial port	USB	Ethernet (10/100 Base T)
Wireless connectivity	CC1000/CC2420	CC1000/CC2420	CC1000/CC2420

Table 2.2: Gateway settings.

MOTE	Wec	René	René 2	Dot	Mica	Mica2Dot	Mica2
Year	1998	1999	2000	2000	2001	2002	2002
MICROCONTROLLER							
Type	AT90LS8535	AT90LS8535	ATmega163	ATmega163	ATmega103	ATmega128L	ATmega128L
Program Memory (KB)	8	8	16	16	128	128	128
RAM (KB)	0.5	0.5	1	1	4	4	4
Active Power (mW)	15	15	15	15	8	8	33
Sleep Power (μW)	45	45	45	45	75	75	75
Wakeup Time (μs)	1000	1000	36	36	180	180	180
NON VOLATILE STORAGE							
Chip	24LC256	24LC256	24LC256	24LC256	AT45DB041	AT45DB041	AT45DB041
Connection Type	I_2C	I_2C	I_2C	I_2C	SPI	SPI	SPI
Size (KB)	32	32	32	32	512	512	512
COMMUNICATION							
Radio	TR1000	TR1000	TR1000	TR1000	TR1000	CC1000	CC1000
Data rate (Kbps)	10	10	10	10	40	38.4	38.4
Modulation type	OOK	OOK	OOK	OOK	ASK	FSK	FSK
Receive power (mW)	9	9	9	9	12	29	29
Transmit power at 0dBm (mW)	36	36	36	36	36	42	42
POWER CONSUMPTION							
Minimum operation (V)	2.7	2.7	2.7	2.7	2.7	2.7	2.7
Total active power (mW)	24	24	24	24	27	44	89
PROGRAMMING AND SENSOR INTERFACE							
Expansion	none	51-pin	51-pin	none	51-pin	18-pin	51-pin
Communication	IEEE 1284 (programming) and RS232 (requires additional hardware)						
BUDGET							
Price (USD)	Discontinued products						155-169

Table 2.3: Crossbow motes evolution since 1998 to 2002.

MOTE	MicaZ	Cricket	TelosB	Imote2	Iris	$\bar{e}Ko$ (eN2100)
Year	2004	2004	2005	2007	2007	2008
MICROCONTROLLER						
Type	ATmega128L	ATmega128L	TI MSP430	PXA271 Intel XScale	ATmega1281	ATmega1281
Program Memory (KB)	128	128	60	-	128	128
RAM (KB)	4	4	10	32	8	8
Active Power (mA)	8	33	1.8	31	8	8
Sleep Power (μA)	20	75	5	390	8	8
Wakeup Time (μs)	180	180	6	-	180	180
NON VOLATILE STORAGE						
Chip	AT45DB41B	AT45DB014B	ST M24M01S	Intel StrataFlash	AT45DB014B	AT45DB014B
Connection Type	SPI	SPI	I_2C	-	SPI	SPI
Size (KB)	512	512	48	32 MB	512	512
RADIO	CC2420	TR1000	CC2420	CC2420	CC2420	CC2420
PROGRAMMING AND SENSOR INTERFACE						
Expansion	51-pin	51-pin	10-pin	10 I/O signals	51-pin	4-ports
Communication	USB	USB	RS-232	USB	USB Client/Host	ESB[2]
BUDGET						
Price (USD)	134	263	134	404	155	3359[3]

Table 2.4: Crossbow motes evolution since 2004 to 2008.

[2]Crossbow ESB (Environmental Sensor Bus) Protocol.

[3]The price corresponds to the commercial kit consisting of three sensing nodes and sensors, an electronic gateway besides the software tools.

2.2.2. Software architecture overview

In this section, the software architecture of a sensor node is explored, from the point of view of application developers. As mentioned, sensor nodes are intended to load and execute a small application, which usually includes the required libraries from an WSN operating system and the application itself. Although the hardware and software boundaries are quite unclear [Hil03], the software architecture of a sensor node could be represented as depicted in Figure 2.7. It shows a layered stack design, where each layer provides its interface to the upper one and uses the interface offered by the one immediately under it.

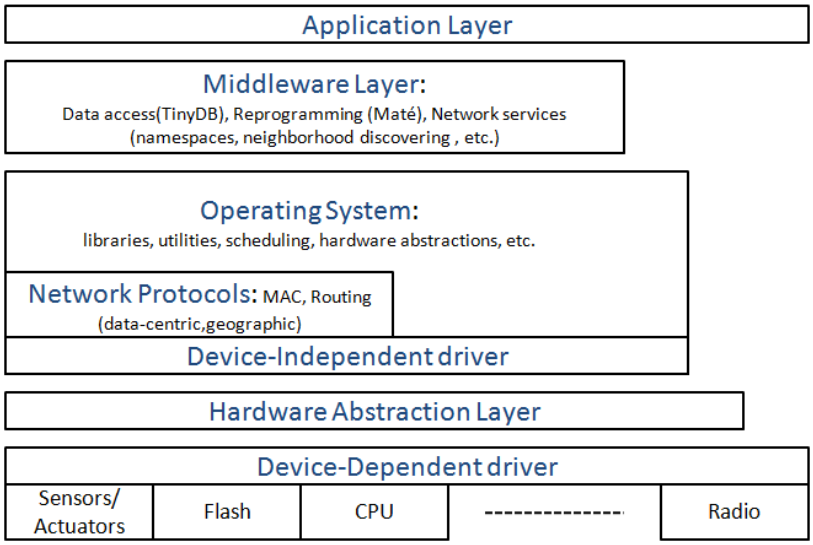

Figure 2.7: Software architecture for sensor nodes.

- In the lowest layer, abstractions of the raw hardware are found, such as proprietary drivers implemented for the physical components. Motes integrate components proceeding from different manufacturers, which must provide their implementation and specification.
- Hardware abstraction layers lie between the previous layer and the operating system. They specify the hardware access performed by the operating system. Given the hardware heterogeneity, HAL aims to homogenize the underlying hardware interface through one generic API, hiding the complexity exposed by physical devices. Unfortunately, this layer is not always present.
- Operating systems (OSes) specifically designed taking into account the requirements and constraints of sensor nodes, manage the complexity of physical components, and provide the necessary abstractions to programmers. Currently, there are several popular WSN OSes such as TinyOS, Contiki, Mantis and LiteOS, as will be discussed in the next sections. OSes use the low-level interface (hardware or HAL) to compose bigger grained operations, which are exported at the upper levels through a well-defined API. In general, OSes cover the required functionality: network protocols (e.g. MAC, routing, discovery, synchronization), local timing, sensing, data storing, power-efficient modes, and so on. Two of them are the most common execution models employed by the WSN OSes: event-driven or threading model, which are discussed in the next section. OSes are multi-platform since they support a wide range of hardware platforms.

- On top of the operating system, a few high-level abstractions can be located such as middleware layers. Due to the fact that resources are limited, OSes do not always integrate the most efficient mechanisms for the whole applications domain. For this reason, middleware approaches have usually been intended to bridge the gap between the OSes and the applications. The goal of these middlewares is very versatile: from network middleware (e.g. reactive middleware to environments and networking conditions such as MiLAN [HMCP04]), development middleware (e.g. Envirotrack [ABC+04]), or middleware to data access (e.g. TinyDB [MFHH05]). Next sections describe some relevant middlewares.
- Finally, in the highest layer of the architecture, applications are found. Applications are programmed respecting the API offered by the underlying operating system, and eventually the middleware layer. However, as previously mentioned, in order to complete applications, usually the programmers are forced to make implementations at different architectural layers.

In spite of a stacked architecture has been described, frequently programmers make use of cross-layer approach, which allows them to directly access any specific component independently of the layer where it is located.

2.2.3. Analysis of operating systems

OSes developed specifically for sensor nodes constitute the backbone of this architecture. The key challenge of the WSN operating systems is to manage the hardware resources of the sensor nodes in an efficient and energy-aware way. Literature on embedded devices reveals the event-based paradigm as the prevailing model, as opposed to the thread-based model. In this section, the two dominant execution models are analyzed and compared, and exponents of each one are described.

2.2.3.1. Event-based vs. thread-based paradigm

Most operating systems designed for sensor nodes can be grouped into one of these two categories: event-based or thread-based. Literature related to embedded systems, and, in particular in WSN, clearly reflects the first one as the dominating approach, due to the hardware constraints.

The event-based model consists of a program which is implemented as a set of independent functions or *event handlers*. Every event handler is triggered as a response to one external or internal event (e.g. hardware interruption). When it happens, the event handler executes atomically (with no interruption) to completion, and finally returns to the caller. In this semantic, there is no possibility of blocking situations while running, and therefore, it eliminates the overhead imposed by the context switching. During its execution, every event handler uses a single memory space allocated for this purpose by the operating system. Due to the fact that the stack is shared, this kind of systems reduces the amount of memory space used, and subsequently, the resources implied in the management. In a thread-based system, the set of actions is performed by execution entities called *threads*, which allocate its own stack. Therefore, context switching and race conditions could happen while a thread is running, which means that inter-thread synchronization and scheduling mechanisms must be provided by the OS. Due to this feature, the event-based model, more lightweight than thread-based, has historically been preferred in embedded devices programming. Figure 2.8 expresses graphically the stack usage in both approaches. In the particular case of WSN, the applications are by nature, reactive and interactive with the environment, and, therefore,

Paradigm	Event-based	Thread-based
Semantic	Run-to-completion	Preemptive
Resources	Shared stack	One stack per thread
Control flow	State machine	Linear
Programming flow	Unstructured, *ad-hoc* code, split-phase functions	Sequential
Synchronization	Asynchronous	Blocking
Concurrency model	Non explicit by event-handlers	Explicitly by threads

Table 2.5: Main characteristics of event-based and thread-based paradigm.

an event-driven model is more convenient, besides that a thread model could result prohibitive given the severe resource constraints.

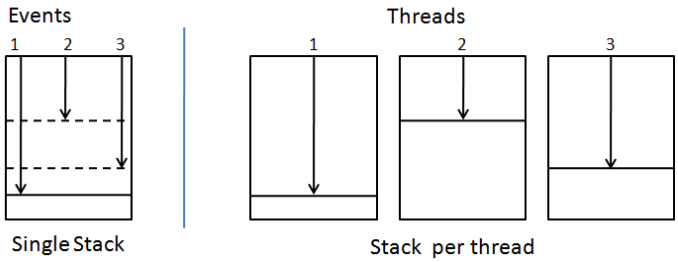

Figure 2.8: Comparison of stack usage in event- and thread-based systems.

In thread-based systems, the parallelism is addressed by the interleaved execution of the scheduling threads in the program. In the event-driven model, the parallelism is not explicit and is carried out by the event handlers.

The event-driven systems programming style is frequently pointed out as a disadvantage. It has often been said that event-driven systems are implemented as complex state machines, where the code is unstructured and non-linear, the control flow is not explicit and the execution order is undefined. Subsequently, event-driven code is hard to read and debug, and, in general, development tools lack mechanisms to explicitly express the program structure. On the contrary, the thread-based programming style is more natural for programmers, since the writing is sequential. Programming languages provide blocking calls to explicitly suspend a thread, which is resumed when blocking conditions disappear. Figure 2.9 compares the control flow in both paradigms. In Table 2.5 the main features of the two models are summarized.

2.2.3.2. TinyOS

TinyOS [Hil03] was the first open source OS specifically designed for wireless sensor devices, developed in the laboratories of the University of Berkeley by Phillip Levis under the supervision of David Culler in 2002. Soon it becomes the *de facto* standard operating system for writing sensor applications. It is also the tiniest of all existing WSN operating systems (a minimum application can occupy around 250 bytes of ROM and 16 bytes of RAM of footprint [HSW+00]). TinyOS is written in nesC [GLvB+03], a component-based programming language based on C that allows programming interfaces and components. nesC distinguishes two kind of components: implementation components, or *modules*, which specify what interfaces use and provide the component, and contain its implementation; and specification components or *configurations*,

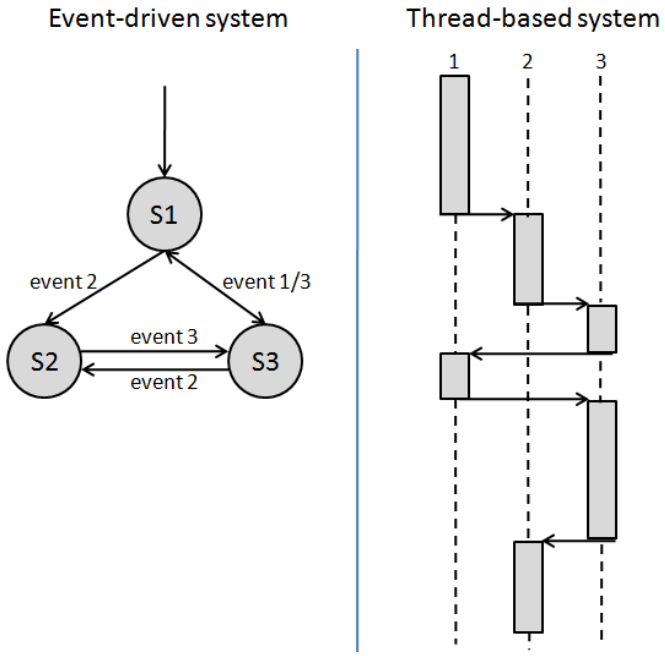

Figure 2.9: Comparison of control flow in event- and thread-based systems.

which describe the relations among interfaces used and provided by components. This paradigm of programming is one of the strong points of TinyOS, due to the modularity of the applications developed. TinyOS also provides flexible abstractions for the different hardware levels in the way of components: from low-level hardware to logic operations in the OS abstraction level.

TinyOS has two scheduling levels: events and tasks. It uses a unique shared stack for both. TinyOS is based on preemptive events[4]. There is also a second level of scheduling: the *tasks*, which can be viewed as non-preemptive functions (of lower priority) that run to completion. Tasks can only be interrupted by events, but not by other tasks. Therefore, scheduling in TinyOS responds to the next algorithm: the TinyOS scheduler manages a FIFO queue with capacity for seven tasks and enables interruptions to receive hardware events. Then, the scheduler goes into a loop waiting for incoming events or tasks. If there are no pending events, the scheduler removes the next task in the queue, and executes it. If an event arrives when one task is being executed, the scheduler interrupts this task, saves its state and executes the associated handler event, and finally restores the execution of the task. Whether, on the contrary, there is a pending event, it first executes the associated handler event, and invokes the task waiting to be processed later.

Among the main disadvantages that TinyOS presents, it is worth mentioning the difficulty of maintaining or updating the application, since it is statically linked to the whole kernel. In this sense, some solutions have been proposed to disseminate the code into the network [LC02, LPCS04, RL03]. On the other hand, the execution model described previously exhibits certain problems related to failures when posting tasks. Since its original version released in 2003, TinyOS has been revised several times thanks to active development groups. The most important revision was in 2006, when the second version of TinyOS (T2) was launched, incorporating meaningful

[4] As opposed to the traditional event-driven paradigm, in TinyOS one event can preempt both events and tasks.

changes which are commented in detail in the next section.

The communication model used in TinyOS is based on *Active Messages*. Active Messages is an asynchronous communication paradigm originally described in [ECGS92] and, subsequently, adapted to the TinyOS network model. Active Messages multiplex the physical radio between different communication channels which are simply labeled with identifiers. Every message incorporates a unique identifier, which is related to the message handler to be invoked when the message arrives to the target node. Message handlers run to completion and its execution is very fast, therefore, it reduces the latency in communications. In this way, Active Messages paradigm fits very well with the asynchronous and event-based execution model of TinyOS.

The second version of TinyOS was released in November of 2006. Besides supporting new hardware platforms, it incorporates multiple improvements over the earlier versions, among which the following are remarkable:

- Hardware access is organized in a *Hardware Abstraction Architecture (HAA)* of three levels (see Figure 2.10):

  1.- *Hardware Independent Layer (HIL)* is the top layer in the architecture, which is independent of the underlying hardware.

  2.- *Hardware Adaptation Layer (HAL)* provides high-level abstractions of the underlying hardware, and therefore, it is platform-specific.

  3.- *Hardware Presentation Layer (HPL)* is the lowest level of the architecture, and it abstracts away the raw hardware presenting it as nesC interfaces.

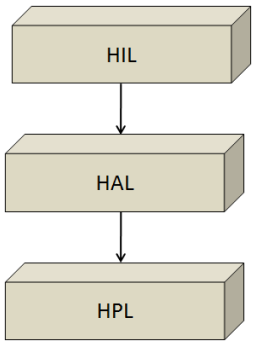

Figure 2.10: *Hardware Abstraction Architecture* described in TinyOS 2.x.

- Scheduler manages, as in TinyOS 1.x, a non-preemptive FIFO queue with capacity for 255 tasks. As opposed to TinyOS 1.x where multiple situations of failure might be produced, *a post of a task only will fail if and only if the task has been previously posted and its execution has not started*. This semantic is achieved by locating a byte of state for every task, to test if it must post itself again or not.

- The boot sequence and initialization is completely different. In TinyOS 1.x, the implementation component was forced to provide the interface *StdControl*, besides the rest of components that need to be powered up. In TinyOS 2.x, that interface is split into two interfaces: *Init* and *StdControl*, which hide the initialization of all internal components related to hardware and operating system. Once it has been completed, the event *Boot.booted* is signaled, whose implementation corresponds to the high-level application.

- Efficient power management is another advantage of TinyOS 2.x. It has taken into account the specific hardware (microcontroller and devices) for energy saving.
- Communication is, as in TinyOS 1.x, based on the Active Messages mechanism. New interfaces, components and data definition have been incorporated, and new network protocols such as *dissemination* and *collection* have been added.

The network model in TinyOS 2.x gives support to 6LoWPAN, the acronym of *IPv6 over Low power Wireless Personal Area Networks*. 6LoWPAN is a working group within the IETF concerned with the specification of transmitting IPv6 packets over IEEE 802.15.4 networks, enabling sensor nodes to take part in the Internet. Up to now, the working group has completed two Internet drafts: *6LoWPANs: Overview, Assumptions, Problem Statement, and Goals* [fC07b] and *Transmission of IPv6 Packets over IEEE 802.15.4 Networks* [fC07c]. Different implementations have been performed in order to be included into the network stack of TinyOS 2.x. Due to the limited size of sensor nodes, these implementations do not fully incorporate the requirements for a full IPv6 stack. For example, the implementation described in [Har07] adapts 6LoWPAN including fragmentation, mesh addressing and broadcast headers, besides compressions mechanisms based on HC1. It includes the ICMP protocol and communication over the UDP protocol.

2.2.3.3. Contiki

Contiki [DGV] is a complete operating system designed for memory constrained systems, developed in Europe in the Swedish Institute of Computer Science (SICS) by Adam Dunkels as the leader of the project in 2003. It has been written in the C programming language, and allows WSN applications to be written whose typical size is around kilobytes (due to the fact that it incorporates new services). It means a bigger footprint than the applications developed in TinyOS. In spite of this, it could be considered the second most extended operating system for programming sensor nodes because it presents some contributions with respect to TinyOS:

- Applications can be more easily updated, due to the fact that Contiki supports load dynamic of programs on the top of the operating system kernel. In this way, code updates can be remotely downloaded into the network. This feature is one of the main advantages of Contiki, given that most operating systems generate an inseparable image of the system.
- Unlike most WSN operating systems, which use an event-driven programming model in order to reduce the overhead of the system, Contiki uses very lightweight threads, called *protothreads* [DSVA06], which can be viewed as blocking event handlers. In this way, light blocking semantic is incorporated for the first time in a WSN operating system.
- It also supports a preemptive multi-thread library in the top of an event-driven kernel. This library can be included on demand by the application.

However, the most relevant contribution has been the use of *protothreads* inside Contiki. Although the protothreads mechanism is not novel and it is based on the concept of *continuations* of Reynolds [Rey93] and Simon Tatham's coroutines [Tat05], they were not incorporated into Contiki operating system until its second version, released in 2005. The Contiki execution model combines events and threads into protothreads, which provide a thread-like programming style (blocking model and sequential flow) and also, just like events, do not require their own stack, reducing the memory overhead. In the words of Adam Dunkels: "*Protothreads are extremely*

lightweight stackless threads designed for severely memory constrained systems, such as small embedded systems or wireless sensor network nodes".

Protothreads provide a set of high-level programming abstractions, which are implemented as C preprocessor macros. Every prothothread is based on a *local continuation*, which locates two bytes to represent the address of jump inside the C function implemented by the protothread. The size of the pointer is platform-dependent, and therefore, the overhead imposed by the use of protothreads will depend on the microcontroller. The local continuation is implicitly updated through these C macros, which basically perform two internal operations: set or resume a local continuation. In this way, blocking sentences can be implemented.

A process in Contiki is a single protothread, which implements a C function. An application can include also several protothreads, which will be executed in the order in which they were scheduled or in the order specified through `AUTOSTART_PROCESSES` statement. The scheduler takes each protothread from a FIFO queue and execute it, until a blocking condition arises. In the hypothetical case that there is no blocking condition, every protothread runs to completion with no possibility of being interrupted by other protothreads. Therefore, the protothreads mechanism does not specify any policy to be scheduled, but the protothreads execution will depend on the order of events execution. In other words, if a protothread is blocked waiting for a condition to become true (for example through `PT_WAIT_UNTIL()` statement), the next protothread that starts to run will be the one associated to the first scheduled event.

Contiki network model provides two independent communication stacks: *uIP* and *Rime*. uIP [Dun03] is based on a lightweight version of TCP/IPv4 stack, and subsequently, providing Internet communication abilities to 8-bit microcontrollers. The basic idea is to remove from the full TCP/IP stack services rarely used by embedded devices: it supports only a single network interface and contains TCP, UDP, IP and ARP protocols. In this way, sensor nodes can partially communicate to other hosts in the Internet. Recently, in November of 2008, IPv6 was released, intended to be the smallest IPv6 stack in the world, with 11 KB of ROM and 2 KB of RAM. Figure 2.11 depicts the Rime stack protocols suite. Rime is intended to offer a set of communication protocols arranged in a layered fashion, where the protocols located in the upper layers make use of the lower layer protocols to carry out their actions. Interactions among Rime protocols is performed via *callbacks*, which means that a certain event is signaled to the upper layer (just as TinyOS events). Nodes using Rime must agree on the channel to use: every channel is identified by a 16-bit number. Rime consists of 16 protocols at different layers, every one of them implementing different routing protocols:

- abc, *Anonymous Best-Effort Single-hop Broadcast*, is the most basic protocol. It sends data to all neighbours in the channel. There is no data about the packet source.
- sabc, *Stubborn Best-Effort Single-hop Broadcast* provides stubborn anonymous best-effort local area broadcast.
- trickle, *Reliable Multi-hop flooding*. It sends a single packet to all nodes on the network.
- uabc (also known as *polite*), *Unique anonymous best-effort single-hop broadcast*, sends one local area broadcast packet within one time interval. There is no data about the packet source.
- ibc, *Identified Best-Effort Single-hop Broadcast*. It sends a packet to all neighbours. The protocol adds the source address to outcoming packets.
- uc, *Best-Effort Single-hop Unicast*. It sends a packet to a single destination in a single-hop.

- uibc (also known as *ipolite*), *Unique identified best-effort single-hop broadcast*, sends one local area broadcast packet within one time interval,
- rudolph1, *Multi-hop Reliable Bulk Transfer*, implements a multi-hop reliable bulk data transfer mechanism.
- rudolph0, *Single-hop Reliable Bulk Transfer*, implements a single-hop reliable bulk data transfer mechanism.
- mh, *Best-Effort Multi-hop Unicast*. It sends a packet to a single destination, using multi-hop forwarding.
- suc, *Stubborn Single-hop Unicast* repeatedly sends a packet to a single-hop neighbour using the unicast primitive (uc).
- nf, *Best-Effort Multi-hop Flooding*. It sends a packet to all nodes in the network.
- ruc, *Reliable Single-hop Unicast*. Reliable single-hop sending of packets to a single destination. It uses ACKs and retransmissions to ensure that the destination has received the packet.
- tree (also known as *collect*), *Hop-by-hop Reliable Data Collection Tree Routing*, implements a hop-by-hop reliable data collection mechanism.
- route-discovery, *Best-effort Route Discovery* does route discovery for Rime protocols.
- mesh, *Hop-by-hop Reliable Mesh Routing*, sends packets using multi-hop routing to a specified receiver somewhere in the network.

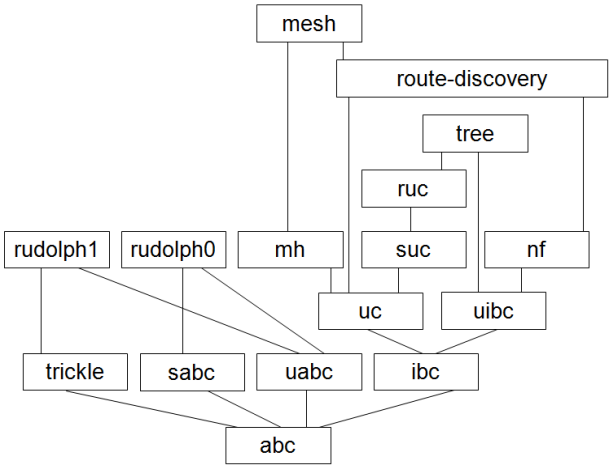

Figure 2.11: Rime stack protocols suite.

2.2.3.4. Other OSes for embedded sensors

Mantis [BCD+05] is a preemptive multithread operating system that was designed for wireless micro sensor platforms at the University of Colorado in 2003. Mantis uses a POSIX-like

programming style, which is one of the strong points of this operating system. However, its footprint is around 500 bytes of RAM and 14 KB of ROM, which means that it is bigger than the footprint of TinyOS and Contiki, since Mantis allocates stack space for every thread (the size of the stack is configurable and ranges between 128 bytes and the maximum available RAM). The Mantis scheduler is based on priorities, therefore, it will execute the highest priority thread to the detriment of threads with lower priority, which might never execute. If a new thread with higher priority than the thread currently executing is scheduled, the first thread will interrupt it, and will start to run. This execution model has certain limitations such as the starvation or the manipulation of the scheduler by the threads with highest priorities.

Sensor Operating System (SOS) [HKS+05] allows applications to be composed dynamically via C modules which are loaded on top of the kernel. SOS modules are independent binaries that implement a determined function. They are not considered processes but they are scheduled cooperatively, which means that they are independent of each other. In this way, one of its main advantages is the dynamic reconfigurability of the system. Since November of 2008 the SOS web site has announced that the operating system *is no longer under active development* and encourages to developers to consider any other WSN operating system more actively supported.

LiteOS [CSAH08] is a recent initiative (2007) for providing a UNIX-like, multithreaded operating system with object-oriented programming support for wireless sensor networks. It includes several features of the Unix systems (e.g. a shell or the programming environment), which increase its footprint leaving it too far from operating systems such as TinyOS. Authors justify this fact by Moore's Law, claiming that in the near future it will be possible to find motes running the Linux operating system.

2.2.4. Communication paradigms, algorithms, and protocols

WSNs have questioned the traditional assumptions related to the communication paradigms employed in conventional computer networks [EGHK99]. In this section, these traditional concepts are reviewed taking into account the new requirements and challenges posed. Under these conditions, the main reference algorithms and protocols are described.

2.2.4.1. Concept review

Communication among sensor nodes is a critical issue for several reasons. On one hand, it is a mandatory functionality since it allows exporting the sensor reads out of the network. On the other hand, this feature supposes the highest energy consumer: the transmission of one single bit consumes the equivalent of 1000 operations of a 32-bit microcontroller [BA06]. Figure 2.12 depicts the energy consumption in the sensor node due to different operations carried out over devices. As shown, the highest consumption proceeds from the radio device, in particular, from transmission operation.

Sensor network applications requirements differ from those ones found in conventional networks such as Internet. The several hardware restrictions make impracticable some conventions employed both in Internet and mobile networks. Naming sensor nodes through IP addresses is avoided: the identification of individual devices cannot always be required [CDE+05], besides being extremely costly in terms of memory. A futuristic approach envisions the existence of inexpensive nodes and large-scale networks, where the ratio between communication nodes and users will be much greater than the existing one between computers and users in the current Internet, and *at such ratios, it is impossible to play special attention to any individual node* [EGHK99].

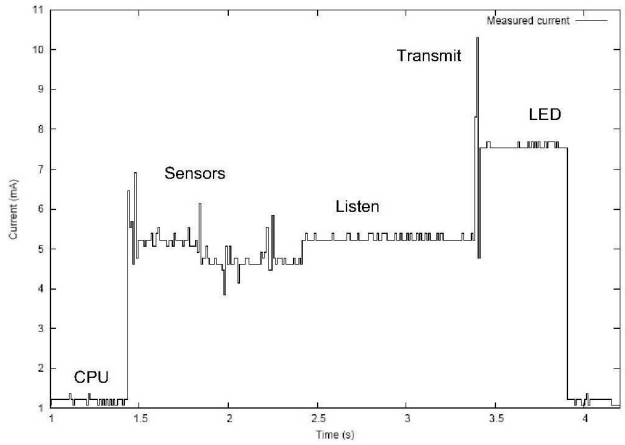

Figure 2.12: Energy consumption in a sensor node due to CPU, sensors, leds, and radio devices.

WSN applications must be designed to be interactive with the environments and to work in an unattended way. They must be reactive to changes and the response time should be short. The network topologies are very dynamic, because of temporarily inactive nodes or its mobility (due to wind or any other climatic conditions). The failures from devices (e.g. sensors may be inaccessible, batteries exhausted, and so on) should be considered as a normal condition, as the WSN should continue performing the global task for which it was created. In other words, algorithms should be robust and stable.

In spite of the fact that more than a hundred wireless standards exist such as Wi-Fi (802.11a/b/g) or Bluetooth (802.15.1), none of them has been intended to completely address the requirements for *Low-Rate Wireless Personal Area Networks* (LR-WPAN) where the WSNs are located. Some characteristics of LR-WPAN are:

- Low-rate data transfer and short distances between nodes. Over-the-air data rates of 250 Kb/s, 40 Kb/s, and 20 Kb/s.
- Low-power consumption.
- Star or peer-to-peer operation.
- Allocated 16 bit short or 64 bit extended addresses.
- Allocation of guaranteed time slots (GTSs).
- Carrier sense multiple access with collision avoidance (CSMA-CA) channel access.
- Fully acknowledged protocol for transfer reliability.
- Energy detection (ED).
- Link quality indication (LQI).
- 16 channels in the 2450 MHz band, 10 channels in the 915 MHz band, and 1 channel in the 868 MHz band.

Standard	Wi-Fi (802.11g)	Wi-Fi (802.11b)	Bluetooth (802.15.1)	ZigBee (802.15.4)
Usage main	WLAN	WLAN	WPAN	Monitoring
Memory	1 MB	1 MB	256 KB	4-32 KB
Lifetime (days)	0.5-5	0.5-5	1-7	100-1000+
Network size	32	32	7	255-65000
Speed (Kbps)	54 Mbps	11 Mbps	720 Kbps	20-250 Kbps
Coverage (meters)	100	100	10 (v.1.1)	1-100
Features	Speed, flexibility	Speed, flexibility	Cost, application profiles	Low-power, low-cost, reliability

Table 2.6: Comparison of wireless standards.

PHY (MHz)	Frequency band (MHz)	Spreading parameters		Data Parameters		
		Chip rate (kchip/s)	Modulation	Bit rate (kb/s)	Symbol rate (ksymbol/s)	Symbols
868/915	868-868.6	300	BPSK	20	20	Binary
	902-928	600	BPSK	40	40	Binary
2450	2400-2483.5	2000	O-QPSK	250	62.5	16-ary Orthogonal

Table 2.7: Frequency bands and data rates.

Consequently, the proposal of a standard considering the particular requirements of WSN became necessary, in order to specify the main directives of actuation for both radio manufacturers and developers. This standard is currently known as *ZigBee*, which is described in the next section. Table 2.6 shows the main features of ZigBee and its comparison to other wireless standards.

2.2.4.2. The 802.15.4 IEEE standard

The *ZigBee Alliance* [CHAj] came about to promote and guarantee the development and implantation of a low-cost wireless technology. It was created for a consortium of semiconductors manufacturer companies (Honeywell, Invensys, Mitsubishi, Motorola, Philips and Samsung) and more than 100 sponsors, with the objective of defining a set of global specifications for wireless applications. ZigBee is also the commercial name given to that specification.

The first version of the 802.15.4 IEEE standard [IEE06] was approved in 2004 to specify the Physical Layer (PHY) and MAC sublayer for Low-Rate Wireless Personal Area Networks (LR-WPAN). The Physical Layer is subdivided into two: PHY data service and PHY management service, intended to specify the service of transmitting the data packet at the physical level, called PPDU (PHY Protocol Data Unit) on the wireless medium. It includes activation and deactivation of radio, channel selection or frequency bands. The standard defines a set of primitives and parameters of configuration, which should be present in any physical protocol. Following the standard, compatible devices should operate in one or several of the frequency bands shown in Table 2.7.

In spite of the fact that ZigBee could use the three frequency bands defined by the physical layer, most radio manufacturers have preferred the 2.4 GHz frequency band because of its global geographical scope. ZigBee focuses on reducing the cost of the radio transceiver more than other wireless standards (in 2006, the price of a ZigBee compatible transceiver was around 1 dollar), and the prevision is that as the technology matures, the cost will be reduced even more. Although

not all motes are compatible with ZigBee and the 802.15.4 standard, the trend is that the new mote models incorporate ZigBee compatible radios (e.g. CC2420) as Tables 2.3 and 2.4 show.

Analogously, the Medium Access Control (MAC) sublayer is subdivided into two parts: MAC data service and MAC management service, which perform the sending and receiving of data units in this layer called MPDU (MAC Protocol Data Unit) through the PHY. Among the services provided we can highlight QoS, reliability mechanism among nodes (ACK), medium access mechanism through CSMA-CA, or security through AES encryption. The advantage of the MAC specification is that it defines only 21 primitives, which results in a simpler hardware that reduces the manufacturing cost.

According to the standard, a ZigBee system is composed of several components. The *device* is the most basic component, and there are two device types participating in a LR-WPAN network:

- *Full-Function Device* (FFD), is a device with capability of communicating to any device, and serving as network coordinator. In accordance with the Authoritative Dictionary of IEEE Standards Terms [IEE00], a *coordinator* is "*A full-function device that is configured to provide synchronization services through the transmission of beacons*[5]. *If a coordinator is the principal controller of a personal area network (PAN), it is called the PAN coordinator*".

- *Reduced-Function Device* (RFD), is a device intended to communicate only to an FFD, and therefore, it can be implemented using minimal resources.

Depending on the characteristics of applications, three different network topologies can be configured, such as Figure 2.13 presents. In the *star* topology, there is a central PAN coordinator, and the communication only can be established between the PAN coordinator and RFD or FFD devices. The PAN coordinator is encouraged to synchronize the communications on the network devices. In the *peer to peer,* or *mesh* topology the communication can occur among any devices of the network, and subsequently more complex network configurations are produced. Sensor networks are a typical example of the use of this topology. Finally, the *cluster-tree* topology can be viewed as a special case of *peer to peer* topology, where FFD devices predominate forming clusters coordinated by a *cluster head* (CLH), and the RFDs can join one cluster at time as a leaf node. Among the CLHs one of them is selected as the PAN coordinator, which is intended to govern the communications in the network.

2.2.4.3. The overlying architecture

The study of energy-efficient network protocols dealing with the main aspects of wireless communication among sensor nodes is currently one of the most important research issues. Power-efficiency is critical. Assuming that the physical layer and MAC sublayer are part of the effort of ZigBee standardization process, this section focuses on describing the upper layers. In terms of ZigBee "*the upper layers consist of a network layer, which provides network configuration, manipulation, and message routing, and, an application layer, which provides the intended function of the device*".

Medium Access Control (MAC) protocols may lie on the MAC sublayer of ZigBee in order to carry out the contention for the wireless media and collisions control. There are many proposed strategies in this area (see Table 2.8), where the main challenge is power saving. Sensor-MAC (S-MAC) [YHE02] has been implemented in TinyOS for Mica, Mica2 and Mica2dot motes (which

[5]Beacon is a special small frame to advertise the presence of a base station.

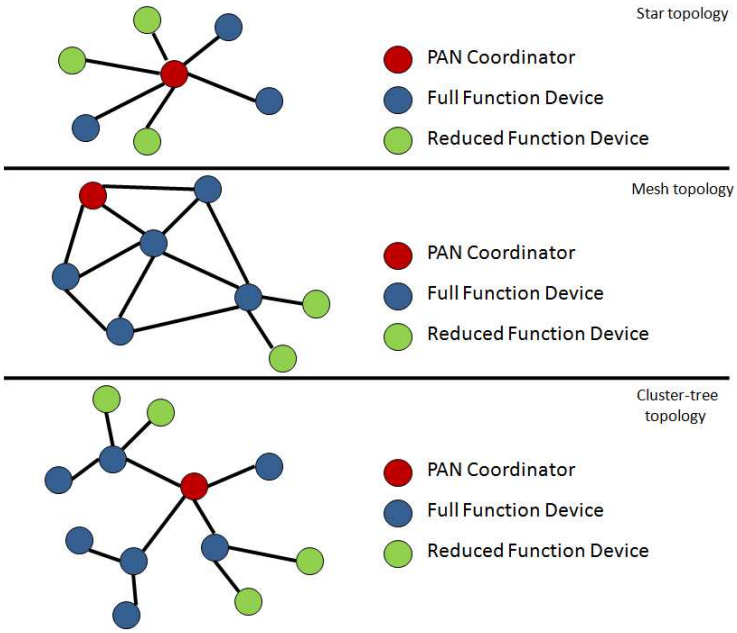

Figure 2.13: The three network topologies of ZigBee: 1) Star topology 2) Peer to Peer or Mesh topology 3) Cluster-Tree Topology.

are not ZigBee compliant), to reduce contention latency for power saving nodes when others transfer data. B-MAC [PHC04] uses adaptive preamble sampling, and supports LPL[6], which addresses the idle listening problem. Both S-MAC and B-MAC are variants of CSMA. There are also schedule-based MAC protocols, most of them a variant of TDMA. These protocols tend to reduce collisions, and therefore they can save more energy. Some examples are TRAMA [ROGLA06], which uses random access signaling slots to exchange neighbor and schedule information, Z-MAC [RWA+08] a hybrid between TDMA and CSMA strategies, and RI-MAC [SGJ08], with random slot assignment in each MAC frame, allows to be mitigated some of the energy wasting problems with respect to other CSMA-based protocols.

Network protocols include *ad-hoc* network discovery, network control and routing, processing information, task and querying and security protocols. The challenges in this area are, besides power saving, robustness and scalability. Routing algorithms are intended to route data from one source to a central node. *Localized algorithms* address the problem of network scalability. The term is described as "*a distributed computation in which sensor nodes only communicate with sensors within some neighborhood, yet the overall computation achieves a desired global objective*" [EGHK99]. Algorithms are based on two principles of design:

- *Data-centric*: Sensor nodes do not need an identifier because applications are not focused on specific devices but on data produced. Questions as: *What is the temperature at sensor X?* are quite improbable, while queries such as *what is the area where the temperature is over 40 degrees?* will be more reasonable.

- *Application-specific*: sensor networks are tailored to the application at hand, in spite of a

[6]In the Low Power Listening (LPL) approach, nodes independently schedule their sleep time, and there is no coordination among nodes.

MAC	Strategy
ALOHA	Originally designed to send packet data over radio networks. It has a simple procedure where data is transmitted and if a collision occurs, wait for random time and re-transmit.
CSMA	*Carrier Sense Medium Access* (CSMA) senses the channel; transmit data only if the channel is free. Collisions can be detected at the receiver.
TDMA	In *Time Division Multiple Access* (TDMA), the channel is split into time-frames which is further subdivided into time-stlots. Each transmitter will be allocated a time-slot during which it can transmit, using the whole channel.
FDMA	*Frequency Division Multiple Access* (FDMA) scheme splits the RF spectrum into fixed number of channels. One of them will be maintained as a signalling channel. A source requests a channel when it has data to transmit.
CDMA	*Code Division Multiple Access* (CDMA) scheme splits the RF spectrum into fixed channels as in FDMA. The use of a channel is based on a pre-assigned hopping sequence, which controls the transmitter by asking it to transmit part of the message on a particular channel, and then hop to another channel.
DFWMAC	*Distributed Foundation Wireless MAC* (DFWMAC) is a four-way handshake mechanism also known as RTS-CTS scheme, which ensures confirmed delivery of data frame.

Table 2.8: MAC Strategies (taken from [Raj05]).

wide range of applications could be implemented.

One example of these kinds of algorithms is *Directed Diffusion* [IGE00]. In Directed Diffusion, all nodes are application-aware, by selecting empirically good paths and by caching and processing data in-network. There are many routing algorithms in accordance with the previous principles of design. *Sensor Protocols for Information via Negotiation* (SPIN) [KHB02] is an adaptive protocol addressing the problem of classic flooding by negotiation and resource adaptation, and subsequently reducing the energy cost. *Low-energy Adaptive Clustering Hierarchy* (LEACH) [HCB00], is a clustering-based routing protocol, which incorporates aggregated data to reduce the amount of data to transmit. Geographic routing algorithms have been proposed to locate sensor nodes with no presence of GPS in each node, and are focused on finding the shortest path. For example, *Greedy Perimeter Stateless Routing* (GPSR) [KK00] uses the positions of routers to make packet forwarding decisions. *Geographic and Energy Aware Routing* (GEAR) [YGE01] selects the most inexpensive neighbor in terms of energy to route a packet through a geographical region. There are algorithms based on AODV (Ad hoc On-Demand Distance Vector) [fC03] or DSR (Dynamic Source Routing) [fC07a] routing, which add certain contributions to extend the lifetime of the network (e.g. turn off the radio when possible). In [Bou09] an interesting categorization of routing protocols is done based on the type of deployment of these networks, which is summarized in Table 2.9.

- *Attribute-based* protocols take routing decisions based on the content of the packets.
- *Hierarchical* protocols create a hierarchy among the nodes in the WSN.
- *Multipath* protocols discover several paths between the source and the destination, and therefore, several alternative routes are ready to be used in presence of failures. In single-path

protocols only one single route is computed. If a failure occurs, then a new route must be discovered via broadcast.

- *Geographical* protocols take routing decisions based on the location devices, in order to estimate the location of a node prior to forwarding the packets to the destination region.
- *QoS* protocols route data taking into account metrics of quality such as energy efficiency to take decisions about routing.
- *Flat* protocols act in flat networks, where there is a large number of nodes collaborating among themselves, and all nodes in the network are equivalent.

2.3. WSN applications programming

This section presents the current state of the art in programming paradigms for WSNs. As previously mentioned, the special requirements of applications in addition to the tight hardware restrictions, make the programming a tricky and error-prone task. For this reason, WSN applications have been traditionally developed *ad-hoc* in a bottom-up approach. This problem represents a relevant challenge for the research community, as is demonstrated by the large number of works on this topic in the WSN literature.

The section starts with a WSN applications taxonomy in order to provide a classification considering certain dimensions and parameters. It establishes a set of requirements to take into account in order to design high-level programming abstractions. Two kinds of paradigms are analyzed: *node-centric* and *macroprogramming*. The first one considers the programming of the sensor node as a whole, while the second approach is focused on describing the distributed processing of the sensor node inside a network. Next, an exhaustive survey of different proposals is presented.

2.3.1. Applications taxonomy

The taxonomy of WSN applications has been examined in different works [RM04, MPar]. The dimensions consider different aspects of applications behavior. As an example, consider Figure 2.14. It shows the taxonomy proposed in [MPar] where the dimensions *goal, interaction pattern, mobility, space* and *time* were identified, and other sub-dimensions are deduced.

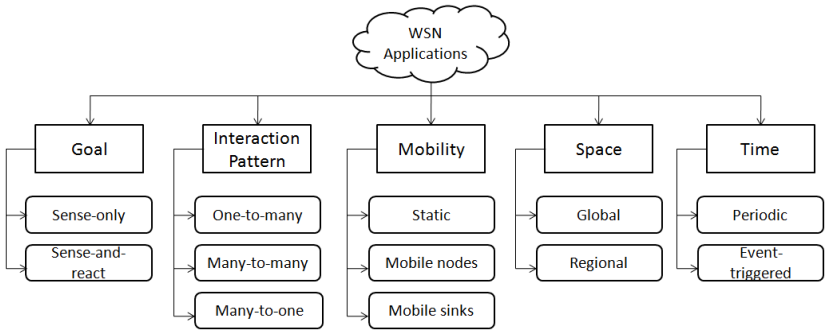

Figure 2.14: A taxonomy of WSN applications (taken from [MPar]).

Protocol	Classification	Mobility	Power Usage	Negotiation based	Data Aggregation	Scalability
Directed Diffusion	Attribute-based	Restricted	Limited	Yes	Yes	Restricted
EAD	Attribute-based	Virtual backbone	Limited	No	Yes	Restricted
RUMOR	Attribute-based	Very restricted	-	No	Yes	High
Youssef et al.	Attribute-based	Cluster based	Low	No	Yes	Limited
LEACH	Hierarchical	Fixed base station	Highest	No	Yes	High
PEGASIS	Hierarchical	Fixed base station	Highest	No	No	High
TEEN	Hierarchical	Fixed base station	Highest	No	Yes	High
APTEEN	Hierarchical	Fixed base station	Highest	No	Yes	High
Al-Karaki et al.	Hierarchical	No	-	Yes	Yes	High
M-MPR	Multipath	No	-	No	No	Restricted
Ganesan et al.	Multipath	No	High	No	No	High
ReInForM	Multipath	Limited	-	No	Yes	Restricted
SPEED	Geographical	No	-	No	No	Restricted
Seada et al.	Geographical	No	High	-	Yes	Restricted
Subramanian et al.	Geographical	No	-	-	No	Restricted
Rao et al.	Geographical	Yes	-	No	-	High
SER	QoS	No	-	Yes	No	Restricted
ARRIVE	QoS	No	-	Yes	Yes	Restricted
SAR	Flat	No	-	Yes	Yes	Restricted
GRAdient	Flat	No	High	No	No	High
MCFA	Flat	No	Low	No	No	High

Table 2.9: Comparison of sensor network routing protocols (taken from [Bou09]).

WSN applications are characterized by their versatility and the great diversity of scenarios where they are deployed. These factors increase the definition of customized requirements and the usage of heterogeneous sensor nodes. Taxonomy allows real world applications to be classified according to the dimensions specified, defining common patterns and thus limiting the design space. It helps developers describe programming models and abstractions covering one or several dimensions. The next subsections explore the different approaches of programming models.

2.3.2. Programming models

Programming models in WSN represent the different proposals to deal with the special features and complexity of the applications programming. Models have focused on two key aspects: *computation* and *communication*, and in general, there is a large void between them. Programming models have eased the programming as a goal, for which they provide abstractions at different architectural levels. According to this classification, two main programming models were defined in the literature:

- The *Node-centric approach* focuses on the programming of individual sensor nodes, without taking into account communication aspects. Hardware abstraction layers and operating systems have been provided. As opposed to traditional systems, the underlying operating system conditions the programming language to be used. Operating systems also present different programming paradigms (imperative, declarative, objects) which increase the heterogeneity and make the portability difficult.
- The *Macro-programming approach* allows the programming of network applications, providing abstractions and constructs to express the distributed processing of sensor nodes. The biggest exponent belonging to this group, is the *middleware*, commonly described to abstract away the programming from the low-level network details and to propose bigger grained operations.

An exhaustive classification is presented in Figure 2.15, which represents a taxonomy of programming models in WSNs [SG08].

2.3.2.1. Node-centric approach

Node-centric programming allows the behavior of individual nodes to be defined, considering them as a whole. WSN operating systems have been designed to write applications on top of them, using the hardware abstractions provided. Although it is possible to obtain a certain hardware independence, operating-system independence has not been completely achieved:

- The programming language for WSN applications building is limited to the used for writing the underlying operating system. Subsequently, applications using TinyOS are forced to be developed in nesC and applications using Contiki, Mantis or LiteOS are developed in C/C++ programming language.
- The execution models and programming paradigm are exported from the underlying operating system to the application level.

Subsequently, WSN applications programming is very close to the operating system, which redounds in a high coding complexity [BP08].

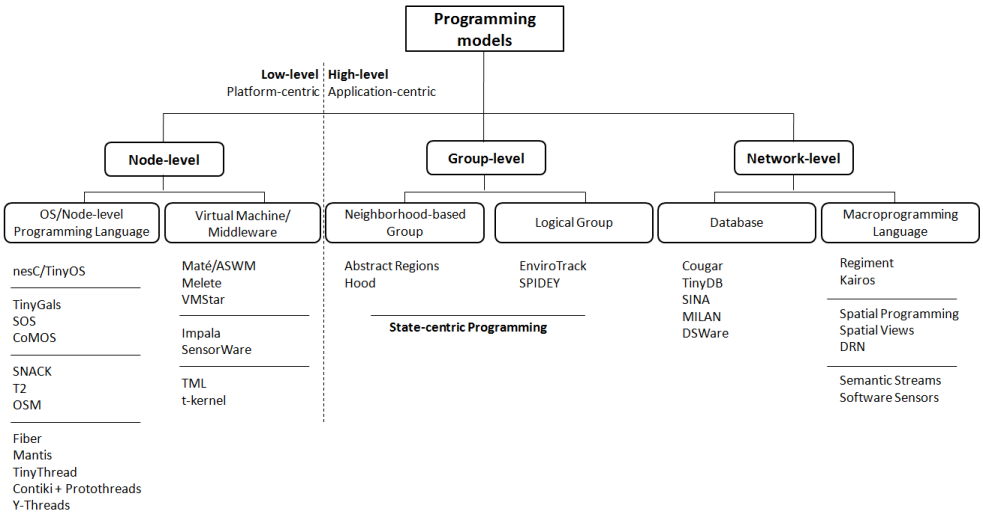

Figure 2.15: Programming models for WSNs (taken from [SG08]).

Due to the novelty of the language, this subsection offers an introduction to nesC [GLvB+03] programming language.

Programming languages: nesC *Network Embedded Systems C* (nesC) is a meta-programming language based on C, aimed at embedded systems that incorporate network management. TinyOS was written in nesC and, subsequently, it is also used to program the user applications developed on this operating system. It supports a programming model that integrates the communication management, concurrency caused by tasks and events, and the ability to react to events which may occur in environments where they are deployed. It also performs optimizations in program compiling, simplifies application development, reduces code size, and eliminates many potential sources of error. Basically nesC offers:

- Separation between construction and composition. There are two types of components in nesC: modules and configurations. The modules provide application code, implementing one or more interfaces. These interfaces are the only access points to the component. The configurations are used to connect the components to each other, connecting interfaces that provide some components with interfaces that others use.
- Bidirectional interfaces: as was explained above interfaces are the components entrances containing commands and events, which are the ones that implement the functions. The supplier implements the commands declared in the interface, while the user implements events.
- Static union components, via their interfaces. This increases the efficiency in execution times, the design robustness, and allows for a better program analysis.
- Several tools that optimize code generation.

Because sensor nodes have a broad range of hardware capabilities, one of the goals of TinyOS is to have a flexible hardware and software boundary. In order to achieve this, nesC components

are abstractions of hardware and software components. However, it is almost always split-phase rather than blocking. It is split-phase in that completion of a request is a call back, and given that components communicate through interfaces, the interfaces are split-phase as well. An important characteristic of split-phase interfaces is that they are bidirectional: there is a down-call to start the operation, and an up-call that signifies the operation is completed.

nesC components can be either *modules* or *configurations*. Modules and configurations differ in the implementation block that follows the specification. The implementation of a module contains the state of the module (variables) and the implementation of every command of the interfaces that it provides, and every event of the interfaces that it uses. This is needed to reach the goal of hiding whether a component is hardware or software. In addition, hardware resources should be *singletons*. Figure 2.16 shows one simple application written in nesC, *Blink*. It distinguishes the configuration (Blink.nc) and implementation (BlinkM.nc) components and the *StdControl* interface provided (StdControl.nc).

Blink.nc (*Configuration* component)

```
configuration Blink{
}
implementation{
  components Main, BlinkM,
             SingleTimer, Leds;

  /* Wirings */
  Main.StdControl → BlinkM.StdControl;
  Main.StdControl → SingleTimer.StdControl;
  BlinkM.Timer → SingleTimer.Timer;
  BlinkM.Leds → LedsC ;
}
```

BlinkM.nc (*Implementation* component)

```
module BlinkM{
  uses {
       interface Timer;
       interface Leds;
  }
  provides {
     interface StdControl;
  }}
implementation{
  command result_t init(){
     call Leds.init();
     return SUCCESS;
  }
  command result_t start(){
     return call Timer.start(
                  TIMER_REPEAT, 1000);
  }
  command result_t stop(){
    return call Timer.stop();
  }
  event result_t Timer.fired() {
     call Leds.redToggle();
     return SUCCESS;
  }}
```

StdControl.nc (*Interface*)

```
interface StdControl{
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

Figure 2.16: A nesC application: *Blink* (written in TinyOS 1.x).

Components can only name variables and functions within their local name space. For one module to be able to use the functionality of another one, it is necessary to map a set of names in one component, generally an interface, to a set of names in another component. In nesC, connecting two components in this way is called *wiring*, and the wirings are implemented into configurations. The implementation of a configuration consists basically in the names of components to be wired and the wiring themselves. Since configurations are still components, a programmer can build an application bottom-up by wiring together simpler components, abstracting from the lower levels of implementation.

Domain Specific Languages (DSLs) As opposed to the general-purpose languages (such as C or Java), a *domain-specific language* (DSL) is a "*programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain*".

In many cases DSLs extend a general-purpose programming language adding domain-specific abstractions to describe the problem better and more precisely, which could require a higher abstraction level. Additionally, "*DSL programs are translated to calls to a common subroutine library and the DSL can be viewed as a means to hide the details of that library*". Subsequently, a DSL compiler or *applications generator* [Cle88] must be provided, in order to translate the high-level code into a low-level implementation. There are many examples of DSLs in different branches of science. Well-known samples are LEX, YACC, SQL or HTML. In the WSN scope, DSLs could contribute notably to fill the gap between domain experts and software developers, improving the mechanisms for requirements specification [Sad07].

Hardware abstraction layers The TinyOS 2.0 approach proposes a division for each hardware platform into a *Hardware Abstraction Architecture* (HAA) [HPH+05]. Its design is organized in three different layers with limited roles and responsibilities. Each level redefines the hardware abstraction offered to the upper layer. These three layers are (from up to bottom): 1) Hardware Independent Layer (HIL) exports a platform-independent interface of applications; 2) Hardware Adaptation Layer (HAL) that represents the intermediate layer where the lowest level interfaces are encapsulated into other components masking the complexity; and 3) Hardware Presentation Layer (HPL) that abstracts away the raw hardware presenting it as nesC structures. HAA does not prevent using cross-layer services, which means that one application could directly access the HAL interface.

μSWN [uSW05] is a Sixth Framework Program Research Project specifying a reusable component middleware for easing the WSN applications development. The middleware lies between WSN operating system and hardware platform, masking the complexity of the hardware layer and offering an homogeneous API for resources management. Currently, μSWN supports TinyOS 1.x and TinyOS 2.x, two real platforms (Tmote SKY/TelosB) and one simulation platform (TOSSIM).

2.3.2.2. Macroprogramming approach

The node-centric approach has certain limitations for programming distributed applications. In this sense, *macroprogramming* or network-level abstractions have as goal "*to realize programming from a macroscopic viewpoint that every node and data can be accessed without considering low-level communications among nodes*" [SG08]. The term can be described as follows: "*programming methodologies for sensor networks that allow the direct specification of aggregate behaviors*".

The most common approach to bridge the gap between the applications and low-level software, has been to develop a *middleware* layer mapping one level into the other. In the following, a brief classification of middleware is described.

Middleware Challenges in middleware for WSNs have been identified in [RKM02]. Each particular solution has been specialized in one or several aspects: reprogrammability, QoS, power saving, adaptability or aggregation functions. Most middlewares direct their effort to provide novel network services which are platform-specific. A short classification of middleware is proposed

in [HMCP04], where they are studied considering their adaptability to the dynamic conditions of the network and other parameters (e.g. code or data management, QoS). In this work the authors argue that there are no approaches providing all the management tools required by sensor network applications, and as an alternative to the existing solutions they present *Middleware Linking Applications and Networks* (MiLAN) [HMCP04]. Another survey of middleware systems is given in [Mar05], which presents the next classification:

- *Classic middlewares*, provide abstractions for communication and application requirements, but in general do not consider security or QoS (e.g. Impala [LSZM04], TinyLime [CGG+05]). TinyLime proposes a middleware approach to facilitate the applications development in mobile *ad-hoc* networks (MANETs).
- *Data-centric middlewares* emphasize in aspects related to data management providing database-like abstractions (e.g. Cougar, TinyDB [MFHH05]). TinyDB is a distributed query processor running on the motes of a sensor network. The TinyDB project focuses on acquisitional query processing techniques.
- *Virtual machines* providing power-aware dynamic reprogrammability of the network application (e.g. Maté [LC02], SensorWare [BHS03]).
- *Adaptive middlewares* to the network and environment conditions (e.g. MILAN [HMCP04], TinyCubus [MLM+05]).

2.3.3. Development and simulation tools

Since the applications programming for WSNs is a tricky error-prone task, several tools have been developed in order to ease the task of building sensor network applications. Each one of these tools provide an *Integrated Development Environment* (IDE) for automating a specific process of applications building. In this section, the state of the art of development and simulation tools is reviewed.

2.3.3.1. Development frameworks

Programming frameworks include development environments of different complexity: from self-contained tools to Eclipse plugins. In this section, the most relevant tools are described.

Viptos *Visual interface between Ptolemy and TinyOS* (Viptos) [CLZ05] is an interface between TinyOS and Ptolemy II [EJL+03]. Viptos is an integrated graphical development and simulation environment for TinyOS-based wireless sensor networks. It allows developers to create block and arrow diagrams to construct TinyOS programs from any standard library of nesC and TinyOS components. The tool automatically transforms the diagram into nesC files that can be compiled and downloaded from within the graphical environment onto any target hardware.

Ptolemy II is a graphical software system for modelling, simulation, and design of concurrent, real-time, embedded systems. Ptolemy II focuses on assembly of concurrent components with well-defined models of computation that govern the interaction between components.

GRATIS *Graphical Development Environment for TinyOS* (GRATIS) [VP02] is a tool that can parse existing nesC files and directories and build-up the corresponding graphical representation.

Consequently, both wirings and code can be generated. The core is an *ANother Tool for Language Recognition* (ANTLR), which is a parser that takes as input existing nesC files and directories and builds-up the corresponding representation in *Generic Modeling Environment* (GME) to be used in the graphical context, or later as libraries in another application development.

Eclipse plugins However, the preferred format of these tools are Eclipse plugins. Several plugins for Eclipse have been proposed for TinyOS applications programming. All of them have similar features, such as syntax highlighting, support for multiple target platforms and TinyOS source trees. TinyDT is a plugin whose internal parser builds an in-memory representation of the actual nesC application which includes component hierarchy, wirings, interfaces and the JavaDoc style nesC documentation. TinyOS plugin for Eclipse provides a zoomable components graph overview like `nesdoc`, a tool included into the nesC compiler to generate documentation for the nesC source files. YETI [BSW06] is another Eclipse plugin, whose parser has been optimized for fast execution of the system.

Others In SNACK [BG04], a construction kit for sensor network applications designed to achieve reusability of applications is presented. PECOS [NAD+02] provides a component-based software development for embedded systems.

2.3.3.2. Simulation tools

Debugging applications for sensor networks is a complex task. The main problem arising from the development environment is not the same as those in which the application will run. Additionally, it is difficult to know with reasonable precision what is happening inside a mote at every moment of the program execution or in the presence of errors. Tools such as visualization are not available. In this section, some of the existing simulation tools have been analyzed.

TOSSIM TOSSIM [LLWC03] is a discrete event simulator integrated into TinyOS 1.x. The principal aim of TOSSIM is the most accurate simulation of applications written for TinyOS. It indeed allows the user to test and analyze the application, correct eventual errors, and run it in a controlled and repeatable environment. This operation is simple because the simulation can be compiled directly from TinyOS code.

It simulates the TinyOS network stack at the bit level, allowing experimentation with low level protocols in addition to top-level application system. The core of the TOSSIM execution model is the event queue. TOSSIM events (which are different from TinyOS events) are generated by the system by commands and TinyOS events. A node identifier is assigned to each event, which allows associating the corresponding action to a node, the instant of time in which the event will be run, and a function that represents the event handler. The events are scheduled in a temporal order: from the earlier to the later. The event handler is typically an interruption handler (in the component hardware abstraction), which signals events and calls TinyOS commands.

The greatest advantage for the programmer is that TOSSIM allows displaying debugging messages inserted into the source code, through the `dbg` function. However TOSSIM is not the best solution in certain situations where it is necessary to measure some real world aspects. Firstly, it is not able to emulate the hardware architecture of motes. For example, a variable of type `int` declared in a TinyOS program, will be physically represented as an integer number in the PC architecture, typically a 32-bit field, which differs from the mote architecture representation.

Secondly, although it provides a scalable and high fidelity simulation of a complete TinyOS sensor network, it has some limitations when phenomena are introduced into the simulation. To solve this problem TinyViz has been implemented, a GUI that permits users to interact with a simulation. However, these interactions are often *ad-hoc*, and they can not reproduce complex scenarios, such as motion. Thirdly, it does not give information about timing and energy consumption because TOSSIM does not model the execution time of the CPU.

Since energy is a critical issue for any sensor network application, and it was not initially considered in TOSSIM, PowerTOSSIM [SHrC+04] arises to be an extension of TOSSIM with the specific goal of power simulating for TinyOS Applications.

The TOSSIM simulator has also been adapted for the second version of TinyOS. It provides two interfaces (one based on Python and the other one in C++) to dynamically interact with the running simulation. Up to now, the only platform supported by this version is MicaZ mote.

Avrora Avrora [Tit05] was created in UCLA Compilers Group in 2004. It is a simulator of programs written for the AVR microcontroller. Currently it is able to emulate the Mica2 and MicaZ sensor nodes. Its flexible and operating-system independent architecture allows the monitoring of programs during their execution. It implements a high-fidelity simulation allowing measurement of mote performance, such as energy consumption and time execution.

Avrora provides a high-resolution simulation with clock-cycle accurate execution. It also emulates the behavior of devices in the microcontroller as well as the behavior of external devices wired to the microcontroller (such as flash memory). Besides, certain environmental conditions can be modeled.

Emulation of sensor nodes behavior and the network at the hardware level, as well as the correct emulation of the program interaction with hardware, allows the study of its behavior at a level that is much closer to reality.

Cooja Sensor networks applications written for Contiki are typically simulated through *Cooja* simulator [VFE+06]. Cooja is a Java application which allows a Contiki application to be loaded into the simulation environment in order to trace and analyze it. It allows setting parameters such as the number of nodes in the network, and the dynamic interaction between the Cooja simulator and every Contiki system. Due to the fact that it is based on Java, different plugins can be written in order to provide new functionality. In the current version, the power profiling for Contiki applications has not been implemented.

2.4. File systems

The continuous data production through a wide set of versatile applications drives researchers to think about different methods of data storing and recovering, which can provide an efficient abstraction to give persistent support to the data generated into the sensor node. Data come from different sources: internal data produced by the sensor node itself, such as sensor measurements or data logs, and external data received from the network, such as data packets traveling through the WSN with many different purposes (e.g. upload applications into sensor nodes or control data). Additionally, as was mentioned in the previous section, the communication cost is high in terms of energy and the environmental conditions or hardware failure might temporarily make the communication inaccessible. For these reasons, different mechanisms of flash-based storing have

been proposed in the literature [KNM95]. In this section, the studies that deal with the problem of local storage in sensor nodes are reviewed.

2.4.1. Matchbox

Matchbox [Gay] was the first file system for sensor nodes. It is implemented for the Mica family motes (Atmel AT45DB041B flash memory chip). The major goals of Matchbox are reliability (detection of data corruption) and low resource consumption. It offers operations for directories and files, which are unstructured and represented simply as a sequence of bytes. Matchbox allows the applications to open different files simultaneously, but it supports only sequential reads and appending writes, and it does not allow random access to files. It also provides a simple wear leveling policies[7]. The Matchbox code size is small, around 10 KB. The minimum footprint is 362 bytes, which is increased when the number of files grows. For each flash memory page a CRC checksum is used to verify the integrity of the file during recovery from a system crash.

2.4.2. ELF

ELF [DNH04] is a file system for WSNs based on the log file system paradigm [KNM95]. The major goals of ELF are memory efficiency, low power operation, and support for common file operations (such as reading and appending data to a file). The data to be stored in files are classified into three categories: data collected from sensors, configuration data and binary program images. The access patterns and reliability requirements of these categories of data is different. Typically, the reliability of sensor data is verified through the CRC checksum mechanism. For binary images a greater reliability may be desirable, such as recovery after a crash. Typically, traditional log-structured file systems group log entries for each write operation into a sequential log. ELF keeps each log entry in a separate log page due to the fact that, if multiple log entries are stored on the same page, an error on this page will destroy all the history saved until that moment. ELF also provides a simple garbage collection mechanism and crash recovery support.

2.4.3. LiteFS

LiteOS, the UNIX-like multi-threaded operating system, includes a built-in hierarchical file system called LiteFS [CA06]. LiteFS supports both file and directory operations, and opened files are kept in RAM. Directory information is stored in the EEPROM while the serial flash stores file metadata. Furthermore, LiteFS implements two wear leveling techniques, one for the EEPROM chip and the other one for the serial flash.

2.4.4. Coffee

Contiki's Flash File System (Coffee) [TDHV09] is the recent file system included in the Contiki operating system. Coffee is a flash-based file system, designed as a combination of extents and *micro log* files. The concept of micro log files is introduced to record file modifications without requiring a high consumption of memory space. In fact, every open file uses a small and constant memory footprint. Other outstanding features of Coffee are garbage collection, wear leveling techniques in order to avoid memory corruption, and fault recovery.

[7]Flash memory pages can only be rewritten a limited number of times, typically about 10000 operations. The techniques dealing with it are denominated *wear leveling*.

2.5. Model Driven Architecture (MDA)

Model Driven Architecture (MDA) [CHA01] is a standard developed by the Object Management Group (OMG) to impulse the usage of models in the process of software development with the goal of achieving portability, interoperability and reusability through architectural separation of concerns. Over the last two decades, OMG has focused on helping to reduce the complexity and costs, standardizing also the Object Request Broker (ORB).

The next sentence clearly reflects the MDA motivation : "*if we built buildings the way we built software, we would be unable to connect them, change them or even redecorate them easily to fit new uses; and worse, they would constantly be falling down*". The proposal of MDA is software development through a model-driven design, where models can help to describe, in addition to data (as XML), the behavior and execution details of platforms. MDA does not only encourage definition models of different system views but also transformation of them: "*There are simply too many platforms in existence, and too many conflicting implementation requirements, to ever agree on a single choice in any of these fields. We must agree to coexist by translation, by agreeing on models and how to translate between them*".

2.5.1. Scope and definitions

The basic idea is to separate the specification of the system from the execution of that system on a given platform. To obtain this division, MDA redefines several concepts, among them:

- Model is *a description or specification of one system and its environment for some certain purpose.* Models can be expressed using a combination of different techniques, such as the Unified Modeling Language (UML) [Gro01], natural or programming languages, or drawings.
- Viewpoint of a system is *a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system.*

MDA defines three viewpoints of a system:

- *Computation Independent Viewpoint* is focused on the requirements of the system and its environment with no knowledge of the system details or platform.
- *Platform Independent Viewpoint* is focused on defining the operation of a system with independence of the specific platform where the system will be installed.
- *Platform Specific Viewpoint* combines the two previous viewpoints, with additional information about the platform and the details of the use of the system.

As a consequence of these three viewpoints, MDA defines one model to represent each one of them. Each model can be expressed in a different way, and the translation between the models should be possible. In this way, a specific system architecture is achieved through a process of transformations of abstract models into specific system designs.

2.5.2. The three MDA models

The three MDA models are: *Computation Independent Model* (CIM), *Platform Independent Model* (PIM) and *Platform-Specific Model* (PSM), which are described in the following paragraphs.

Computation Independent Model (CIM). CIM is the model representing the Computation Independent Viewpoint. Frequently, CIM collects the system domain and the vocabulary used by their practitioners, and plays the role of intermediary between experts in the domain and experts in software development. For this proposal *Domain-Specific Languages* (DSLs) can be employed.

Platform Independent Model (PIM). PIM is the model representing the Platform Independent Viewpoint. The key idea is that the operation of the system can be specified with a high degree of independence from the platform where it will be installed. Moreover, one single PIM should be valid for different platforms.

Platform-Specific Model (PSM). PSM is the model representing the Platform Specific Viewpoint. It combines the specifications of the PIM with the details of usage a particular platform. In addition, MDA also defines a *Platform model* as the set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform.

MDA does not define how the transformation among models should be done, but the process to perform. This process includes several stages, among the most relevant are:

- Specify the CIM, PIM and PSM.
- Define the *MDA mappings*, or the specifications for transformation of a PIM into a PSM for a particular platform. It could use meta-models, templates or profiles.
- Carry out the *Model transformation*, which is intended to convert one model to another model of the same system. The input to the transformation is the marked PIM and the mapping, and the output is the PSM and the record of transformation. The process of transformation could be both manual and automatic.
- Generate deployable code directly from PIM through a tool created for this purpose.

MDA has been successfully used in the building of complex distributed systems [GBK+05]. Translating the lessons learned in the WSN area, is still a research challenge.

With the emphasis on increasing the software productivity, other strategies can be found, such as *Software Production Line* (SPL). According to *Software Engineering Institute* (SEI), a *software product line* (SPL) is "*a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*".

2.6. Chapter summary

In this chapter, the general concepts of wireless sensor networks have been presented. Hardware and software technology of sensor nodes have also been described. In particular, WSN operating systems have been extensively analyzed and compared in order to identify the advantages and disadvantages of each one. Table 2.10 shows a summary of the operating systems studied in this chapter. The special features of the communication have also been described, paying special attention to the 802.15.4 IEEE standard and ZigBee. Table 2.11 summarizes the file systems studied in this chapter. Table 2.12 shows a summary of the visual tools for graphical composition of applications and their major features. The debugging problem of sensor networks applications has

been identified. The phrase "*there is a tremendous gap between it works in the lab and it works in the real world*" from Matt Welsh describes this situation. Different simulation tools have been presented, some of them are operating system-dependent such as TOSSIM (TinyOS) or Cooja (Contiki). Finally, an overview of the MDA standard is provided in order to present the more relevant concepts, which are used in the following chapters.

OS	Year	Execution Model	Programming Language	Code Magnitude	Dynamic load	Component based
TinyOS	2002	Events	nesC	Bytes Order	No	Yes
Mantis	2003	Multithread	C	KB Order	Yes	No
Contiki	2003	Protothreads	C	KB Order	Yes	No
SOS	2005	Modules	C	KB Order	Yes	No
LiteOS	2007	Multithread	C++/C	KB Order	Yes	No

Table 2.10: Comparison of operating systems for sensor nodes.

Feature	ELF	Matchbox	LiteFS
1	TinyOS	TinyOS	LiteOS
2	Mica2	Mica Family Motes	MicaZ
3	Dynamic	Static	Dynamic
4	RAM, EEPROM, Flash	Flash	RAM, EEPROM, Flash
5	14 bytes (per flash page) 14 bytes 14 bytes per i-node (RAM)	8 bytes (per flash page)	8 bytes (per flash page) 168 bytes RAM 2080 bytes ROM
6	Unlimited	2 (Read/Write)	8
7	Sensor Data Configuration Data Binary program Image	Data files	Data Binary applications Device Drivers

Table 2.11: Comparison among file systems. Features: 1:Operating system; 2:Sensor platforms; 3:Memory allocation; 4:Memory chips used; 5:Metadata size; 6:Number of files opened; 7:Types of files.

Tool	Platform	TinyOS Source Tree	Programming Language	Simulation Environment	Code Generation	Compilation & Deployment
GRATIS(2003)	MS Windows	1.x	C/C++	No	Yes[8]	No
Viptos(2006)	MS Windows Unix/Linux	1.x	Java	Yes	Yes[8]	Yes
TinyDT(2005)	MS Windows Unix/Linux Mac OS	1.1.0/14	Java	No	No	No
TinyOS Plugin 2005	MS Windows Unix/Linux Mac OS	1.15	Java	No	No	No
YETI(2006)	MS Windows	1.1.15	Java	No	No	Yes

Table 2.12: Comparison of tools for graphical composition of TinyOS applications.

[8]Configuration component and Makefile.

Chapter 3

Problem statement

Embedded systems exhibit certain features conditioning the way in which these devices are programmed. Historically, the applications written for these devices have been developed in an *ad-hoc* way, which means that there is a strong coupling between the applications and the platform where they will be deployed. Specifically, sensor nodes are still characterized by scarce resources compared to other embedded systems, and subsequently, the available hardware budget is more restricted. Locating a WSN-specific operating system on top of the hardware abstractions hides the hardware complexity and manages the resources, but does not completely eliminates the coupling of applications.

The goal of the current chapter is twofold: on the one hand, the applications programming problem in the wireless sensor networks scope is presented, and their major causes and consequences determined; on the other hand, an overview of the solution proposed in this thesis is presented as an alternative to address the aforementioned problematic.

3.1. Problem analysis

WSN applications programming is a known challenge in the research community. Several works have studied different aspects of the problem in depth: from determining the WSN applications taxonomy to provide high-level programming abstractions which are usually specific for a type of applications [MPar, SG08].

The current programming of WSN applications requires knowledge of the low-level details, deriving in *ad-hoc* developments of limited reusability. This fact has attenuated the potential benefits of this technology over multidisciplinary areas as was envisioned in [LL05, CDE+05]. In this thesis, three problem sources are identified:

- The complexity inherent to hardware, with an incessant increment of new heterogeneous platforms, a changing hardware, miniaturization research and strong constraints of resources.
- Portability, because WSN operating systems do not always provide the abstractions which allow a platform-independent design and implementation. Moreover, not all sensor operating systems support hardware diversity [Man07], which implies an heterogeneous connections map between the hardware and the operating system.
- Lack of specific standards, programming interfaces and open software architectures governing the development process, and clearly establishing the main directives for developers.

In the following sections of this chapter these three problem sources are analyzed in depth. Subsequently, consequences are presented, and finally, a hypothesis which provides a solution is established for this thesis.

3.1.1. Hardware complexity

Hardware manufacturers are continuously researching new hardware platforms for sensor nodes, incorporating requirements such as power-efficiency microcontrollers, ZigBee compliant radios, innovative sensors or alternative batteries enlarging the sensor node lifetime (e.g. *scavengers*). The spectrum of possibilities is high: until now more than thirty commercial platforms (according to [AS08]) have been discovered (besides proprietary ones). Such diversity has been partially favored by the wide range of applications requirements [RM04]. Hardware complexity can be broken down according to the following factors:

- The rapid growth of both proprietary and commercial hardware platforms.
- The heterogeneity of the devices hold in the sensor node, such as microcontrollers, sensors, or radios, which also present very different settings and features.
- Miniaturization frequently implies more restricted capabilities (memory, CPU cycles, data rate).

Figure 3.1 shows the evolution of the number of sensor nodes manufactured in the last decade. The platforms considered in this study are the following: Wec (1998), Rene (1999), Rene2, and Dot (2000), Mica2 (2001), Mica2dot (2002), Eyes, Nymph, and IRIS (2003), MicaZ, Cricket, Bean, GWNODE, and Mulle (2004), TelosB, BTnode, TinyNode, Tmote SKY, XYZ, FireFly, ESB, and SUN SPOT (2005), Flat Mesh 1, and Flat Mesh 2 (2006), Imote2, CargoNET, SenseNode and Kmote (2007), $\overline{e}Ko$, MSB, Jcreate and SHIMMER (2008).

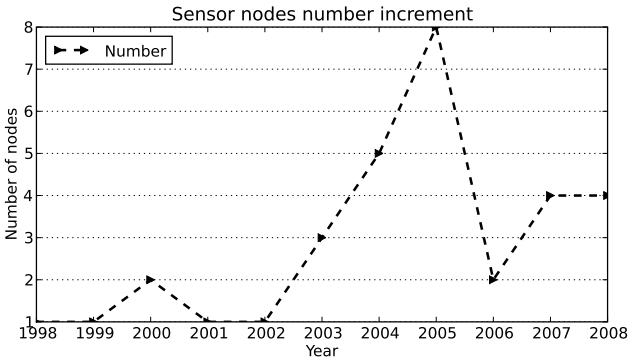

Figure 3.1: Hardware platforms evolution.

These factors affect the overlaying software, determining the maximum amount of available space that applications can allocate (data and code). Programming implies a budget-aware development where the developer must be conscious of both functional and non-functional requirements. Additionally, in order to be exploited, a platform requires being supported by a WSN operating system, and currently there is no operating system covering the whole set of existing hardware platforms.

3.1.2. Applications portability

Portability can be defined as *the degree of independence that applications present with respect to the execution platform*. In some systems, software cannot be reused among different platforms. Thus, distinct platform-dependent applications must be implemented, in order to make the software executable.

WSN applications are built on top of a software stack including a general-purpose operating system (see Figure 2.7). Such an operating system has been specifically designed taking into account the special features of sensor nodes, and is mainly intended to manage the hardware resources and to provide programming abstractions. Nevertheless, they do not completely hide either the underlying platform or its execution model, making the applications programming close to operating system and thus reducing the portability.

In traditional systems, such independence has been generally achieved through high-level abstractions such as POSIX [IEE96], basically consisting of generic interfaces which can be mapped into more specific calls. In general, the cost of achieving portability has been paid in terms of efficiency (more resources consumption). However, in embedded systems, where hardware capabilities are scarce, incorporating some overhead to the applications might be a non-feasible solution. In this subsection, the portability offered by the WSN operating system and hardware is studied.

3.1.2.1. Portability at OS level

Operating systems, traditionally located between applications and hardware in conventional systems, are intended to orchestrate the vertical access among different architectural layers which are arranged in an ascendent way, from the hardware to the application at different abstraction levels. Applications are hardware-independent because the operating system provides abstractions to hide the low-level details of the devices and their management. This architectural division benefits applications portability, ease of programming, and a simple and clear design that limits the responsibilities at every abstraction level. In the particular case of embedded systems, such as sensor nodes, the operating system has successfully taken on its role as an integrator element of versatile applications, hardware and the operating system itself. However, due to the diversity of the hardware, not all the platforms are supported by the whole set of WSN operating systems, as Table 3.1 shows. The effort of porting to new platforms varies depending on the novelty of the platform itself, and usually implies the work of hardware experts and software developers.

Networked sensor operating systems have not been conceived to satisfy some relevant challenges:

- The abstraction level of applications with respect to the underlying levels is very low because the programmers must explicitly select the hardware and software components required by the applications. This task can be only accomplished by expert developers, and it forces an in-depth exploration of all potential alternatives.
- There is no clear division in architectural layers with standardized interfaces, separating the different abstraction levels. Subsequently, there are no established roles for each layer which can limit responsibilities. Moreover, the boundary between hardware and software is still an active research area [CDE+05, LC02].
- In addition to the above, the access to the hardware is arbitrary and issues, such as access patterns or protection mechanisms, have not been taken into account.

These challenges have been, in part, increased by the tight hardware restrictions of sensor nodes, which limit the expected generality at this abstraction level to provide applications portability. Therefore, operating systems must achieve a compromise solution among the available resources and the expected functionality for a general-purpose operating system.

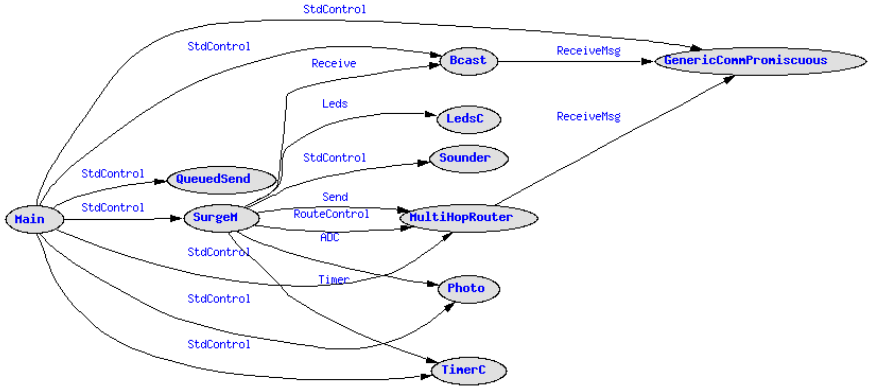

Figure 3.2: *Surge* application components graph.

Consider the TinyOS application represented in Figure 3.2. This shows the component graph of the *Surge*[1] application, generated by the `nesdoc` tool. Every node in the graph is a TinyOS component and every link represents a directed interface between the user and the provider. As can be observed, the system architecture is not clearly defined and it is unknown the layer where the components are located, which causes several problems: ambiguous and complex design, access order among components is not previously specified because all the components can potentially connect to other ones (each one of these connections must be made explicitly). The *Surge* application representation shown in Figure 3.3 looks more suitable, where every node accessed by the application component groups a conceptual functionality: network, I/O, timing and scheduling services.

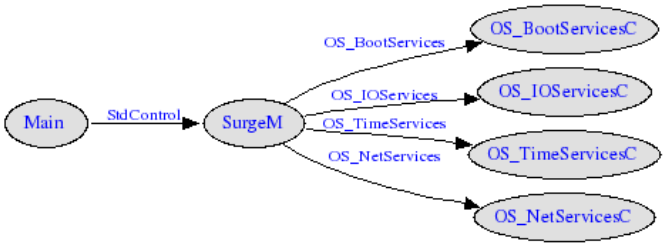

Figure 3.3: A more suitable representation of *Surge* application. It shows a conceptual division of the functionality, and defines the access order between the application and operating system level. The rest of the operating system components are kept behind the four main components: OS_IOServicesC, OS_NetServicesC, OS_TimeServicesC and OS_BootServicesC.

Additionally, the task of orchestrating interfaces and components is not trivial when the application volume grows. The problem persists in other non-component-based operating systems, e.g. Mantis, where the abstraction level is similar: programmers must always specify which hard-

[1] *Surge* is an application to monitor some sensors periodically and send out the data through the radio device.

ware components are used by the application. Therefore, a WSN application programmer must be able to:

- Know in depth the physical devices integrated into the sensor nodes for which the application is being developed.
- Name explicitly in their applications the part of the operating system (components, services, header files) providing the software functionality required. To do this, programmers must be able to select the most suitable components and interfaces, among the whole set offered. This task requires an exhaustive study of the operating system.
- Specify the hardware abstractions that can map the devices integrated into the sensor nodes where the application will be executed. These hardware abstractions could differ among platforms.

Other important aspects to mention are the restrictions imposed by the operating system itself on the applications. As was previously stated, the software for sensor nodes has been traditionally developed *ad-hoc*. That means that applications are tailored to a specific hardware and operating system. Subsequently, in many cases portability and interoperability have been sacrificed, with a strong coupling existing between the application and the underlying operating system.

The operating system determines the way in which applications will be programmed. Frequently, the same application looks very different depending on whether it has been programmed on Contiki, TinyOS or other, even for simple applications with very few lines of code. Specifically, the WSN operating system directly affects applications portability in the following way:

- Applications must be written using the same programming language as the underlying operating system. For example, TinyOS applications must be written in nesC, and Mantis and Contiki applications should be written using the `C` programming language.
- There are no standardized interfaces. A WSN application uses the interface published by the underlying operating system, which should export both hardware and software interface. Eventually, the application could access the hardware interface directly (cross-layer services).
- There are no tools between the application and operating system level such as metalanguages or high-level abstraction layers, with the aim of easing programming. Moreover, there is a complete lack of open programming standards.
- There is a strong coupling between the programming style and its execution model. In fact, the execution model is not kept hidden and it determines the programming model. This fact complicates even more the programming task adding greater complexity and heterogeneity (see Table 3.2).

Therefore, writing operating system-independent applications is still a challenging task, mainly due to the particular features of sensor nodes and the restrictions imposed by the operating systems. Subsequently, WSN applications portability is only partially achieved, as in the different works in the literature pointed out [Lev06]. As shown, writing applications is an error-prone, complex and tricky task.

Other restrictions are given by the features of the operating system itself. A single application can run into the microcontroller (*mono process*), and, consequently, there is a single execution

TinyOS	Contiki	Mantis	SOS	LiteOS	Other
Mica	Tmote Sky	Mica2	Mica2	MicaZ	Eyes (PeerOS)
Mica2	JCreate	Mica2Dot	MicaZ	Iris	Bean (YATOS)
Mica2Dot	TelosB	MicaZ	XYZ mote		GWnode (Proprietary)
MicaZ	Atmel Raven	TelosB	Tmote Sky		FireFly (Nano-RK)
TelosA	MSB	MANTIS nymph	Kmote		SUN SPOT (Squawk)
TelosB	ESB	Tmote SKY	TelosB		FlashMet
BTnode3	MicaZ		Cricket		CargoNET
Rene					SHIMMER
Eyes					
Imote					
Imote2					
Cricket					
Tmote Sky					
TinyNode 584					
Mulle					
SenseNode					
$\overline{e}Ko$					

Table 3.1: Supported hardware platforms (to date).

thread. However, typically applications are also based on events, which can be signaled through hardware interruptions at any given moment of the execution. This feature creates race conditions and concurrency problems, which should be managed by the programmers. Scheduling is another aspect to consider. In TinyOS, events have the highest priority, while Mantis, for instance, allows assigning different priorities to processes. Memory management performed by the operating system also affects the programming. In TinyOS 1.x, the allocation of memory is static, which prohibits the use of function pointers, and also forces to know the exact size of variables at compilation time.

Applications portability at the operating system level is also one of the hot topics for WSN developers community, as is demonstrated by the high number of messages dedicated to this subject in the mailing list of the most important OSes.

3.1.2.2. Portability at HW level

As shown in the previous subsection, the operating system is supposed to efficiently manage the hardware resources of sensor nodes. In general, operating systems model the hardware as a software abstraction, masking the hardware heterogeneity. A WSN operating system also exports several interfaces of hardware resources at different abstraction levels. For example, TinyOS 1.x classifies the components into three categories depending on the abstraction level: hardware abstractions, synthetic hardware, and high level components. However, in TinyOS 2.x (see Section 2.2.3) the hardware architecture is organized in three levels. In spite of the existing artificial divisions, the application components can usually access directly to whatever hardware components. It has been traditionally denominated the *cross layer* approach.

Typically, physical component manufacturers provide the lowest level drivers of the devices. A team of developers integrates the drivers into the corresponding operating system, implementing the hardware components, which can be incorporated into future releases. This means that, low-level abstractions are not always available (in terms of TinyOS 1.x), or in other words, the

operating system does not always provide support for devices. In that case, many programmer teams opt for developing the required software by themselves. Obviously, when a new sensor node platform appears, the current version of the operating system could not support it. In fact, TinyOS 2.x has been ported to many different platforms, which are not supported by TinyOS 1.x.

Depending on the operating system, the effort of porting to a platform varies in terms of complexity, which increases the development times. In general, the task of porting to a new platform is tricky, requiring developers to be able to program at the hardware level. Next case study gives some instructions for porting a new hardware platform, which are based on the Contiki operating system (extracted from [Dun07a]).

1.- Contiki has been designed to be easy to port. The directory structure of Contiki distinguishes the next main locations:

   - The `cpu` directory contains the common code to all platforms, which have the same microcontroller. A subdirectory must exist for each different microcontroller.
   - The `platform` directory contains the specific-platform code. A subdirectory must exist for each different sensor node platform. A template is available in the `native` subdirectory, which can be used for porting new platforms. The `native` subdirectory contains several files which should be customized for each platform: clock implementation, configuration and compilation options, and device drivers such as network or sensor drivers.

2.- The clock implementation implies rewriting the `clock.c` file, and, in particular, three functions: `clock_init.c`, `clock_time()`, and `clock_delay()`. Therefore, the implementation code is architecture-dependent but uses a generic interface exported to all implementations. For example, the next fragment of code shows a possible implementation for the `clock_delay()` function, which delays the CPU for a multiple of 2.83μseconds.

```
void clock_delay (unsigned int i) {
    asm("add#-1,r15");
    asm("jnz -2");
}
```

3.- Device drivers such as network, serial port and sensor drivers must be rewritten keeping in mind some recommendations about the interface which must be respected. The source code is located inside the `net` and `dev` subdirectories.

4.- The configuration and compilation options are grouped in the `contiki-conf.h` file. This includes platform specific settings such as the C compiler configuration, C types, and clock configuration. The applications building system incorporates several `Makefile` files which must be updated in order to specify source files and directories. The building system must also be linked to the Contiki build system in order to compose the complete executable code.

5.- Only if the new platform integrates a new microcontroller is necessary to also port the multithreading library and the ELF loader.

3.1.3. Lack of software architectures

As shown in Section 2.2.2, from the point of view of the software architecture, there are several abstraction levels: hardware, operating system, middleware, and application. For each level, different working areas have been identified, where the programmers usually make the greatest effort of programming. Moreover, invoking hardware components directly from application modules is an accepted programming technique. Subsequently, the work of applications programmers do not consist only in the applications building itself, but, additionally, in the development at the underlying levels: device drivers, network protocols, tailored software components, or distributing one application into different tasks to be performed in cooperation among different sensor nodes [CLZ05]. Software should also be robust enough, stable, and implementing aspects about security, network reconfiguration, or reprogrammability. In this sense, the trend has been the construction of middlewares to support the development process. However, the middleware solution has usually contributed to addressing different requirements in a particular scenario, but it lacks the completeness and generality supposed at this abstraction level.

An architecture is defined as (according to IEEE Std 1471-2000 [Gro00]): "*the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution*". The study of WSN architectures is a challenge, as several authors have pointed out in the literature [CDE+05, LL05, HKKW03]. In the words of David Culler: "*the primary factor currently limiting progress in sensornets is not any specific technical challenge (though many remain, and deserve much further study) but is instead the lack of an overall sensor network architecture*".

The presence of a generic software architecture for WSN should greatly support applications programming. Making a platform-independent design would contribute to achieving reusable and easily understandable solutions. Additionally, a software development methodology could set the basis for the building process, establishing the life cycle, its development phases and interactions. It should improve the development of architecture-independent applications, contributing more simple programming and, consequently, extending the WSN development to a community of non-expert programmers.

3.1.4. Samples and evidences

Samples of the problems referred previously are presented in order to show some evidence of the aspects described earlier in this chapter. First, we compare the most basic application in the WSN scope for different operating systems, i.e., *Blink* application. Later, an example of a non-portable application is analyzed.

3.1.4.1. The *Blink* application

Its goal is to display a base-8 counter using the three LEDs of the mote. This application has been modified to look even more simple: just when the timer fires, the red led will toggle. The application has been developed for the TinyOS, Contiki and Mantis operating systems[2]. Table 3.2 shows the result. Even for the most basic application (it simply consists of two basic operations: start a timer and toggle a led), the execution model code is completely coupled to the application level. Moreover, most of the code corresponds to the execution model abstractions. Note that, for

[2]For the TinyOS applications, only the *module* component is shown, while the configuration component has been omitted.

different execution models, the corresponding programming style is used, with each one of them being very different.

3.1.4.2. *Surge* application: a non-portable application

Depending on the platform for which the application is programmed, the same application could require different implementations. One example is the *Surge* application. *Surge* was initially designed for the first Mica motes and, later, modified for Telos motes.

Since Telos and MicaZ motes incorporated the new CC2400 radio (while Mica2 motes were not compatible with the 802.15.4 standard) several independent network modules were implemented, leading to a new application called *SurgeTelos*. The new implementation presented a signature and interface different than the counterparts in the *Surge* application. Obviously, both applications used different protocol stacks but, at the application level, the main difference lies in the network components listed in the configuration component, because the name provided for each one of the network components is different. In other words, a homogeneous name space for TinyOS 1.x applications has not been exported.

3.2. Consequences and challenges

The immediate consequences deriving from the problems described above are related to three main topics: applications development, portability and usability.

The applications development process itself constitutes an additional obstacle. This process can be characterized in the following way:

- *Ad-hoc* and bottom-up programming. Applications building is frequently close both hardware and operating system, which complicates the process and makes the programming unclear and tricky.
- Complex and platform-specific design limit reusable solutions.
- Reduced interoperability and reusability. Since the software is platform-dependent, in some cases the applications will not be able to communicate or interoperate.
- Carried out by expert programmers, who will have to be concerned with the low-level details of the platform on which the application is going to be installed. Programmers must keep in mind the available resources, implying an additional effort.

Portability Applications frequently are platform-specific, which means that portability of applications across different platforms is not always possible with no changes. Thus, specific implementations in order to port a part or the whole application can be necessary to make the program executable on different platforms. As consequence of it, the effort of programming gets still more complicated, involving longer development times and penalizing the *productivity*. Productivity can be understood as the efficiency indicator that relates the amount of product obtained and the resources employed.

Blink for TinyOS 1.x

```
module BlinkM {
  provides {
   interface StdControl;
  }
  uses {
   interface Timer;
   interface Leds;
  }
}
implementation {
  command result_t StdControl.init() {
   call Leds.init();
   return SUCCESS;
  }
  command result_t StdControl.start() {
   return call Timer.start(TIMER_REPEAT
     , 1000);
  }
  command result_t StdControl.stop() {
   return call Timer.stop();
  }
  task void taskToggle() {
   call Leds.redToggle();
  }
  event result_t Timer.fired() {
   post taskToggle();
   return SUCCESS;
  }
}
```

Blink for TinyOS 2.x

```
module BlinkC @safe()} {
  uses {
   interface Leds;
   interface Boot;
   interface Timer<TMilli> as Timer0;
  }
implementation {
  event void Boot.booted(){
   call Timer0.startPeriodic(1000);
  }
  event void Timer0.fired(){
   call Leds.led0Toggle();
  }
}
```

Blink for Contiki

```
PROCESS(blinker_process, "Blinker");
AUTOSTART_PROCESSES(&blinker_process);
PROCESS_THREAD(blinker_process, ev, data) {
  static struct etimer etimer;
  PROCESS_BEGIN();
  etimer_set(&etimer, CLOCK_SECOND * 1);
  while(1) {
   PROCESS_WAIT_EVENT();
   if(ev == PROCESS_EVENT_TIMER) {
    if(data == &etimer){
     etimer_set(&etimer, CLOCK_SECOND * 1);
     leds_toggle(LEDS_RED);
    }
   }
  }
}
```

Blink for Mantis

```
void blink_thread(void){
  while(1){
   mos_led_toggle(0);
   mos_thread_sleep(1000);
  }
}
void start(void) {
  mos_thread_new(blink_thread,
   128, PRIORITY_NORMAL);
}
```

Table 3.2: *Blink* application for different operating systems. It toggles the red led every 1000 milliseconds. Due to the different execution models and the programming model, the application looks very different.

Usability can be understood as *the capability of a system to be used*. Usability in WSN constitutes an important challenge: while technological maturity is being achieved and resources are available, making it widespread to the average person who can exploit results which are currently unthinkable. In fact, MIT researchers Joshua Lifton and Mathew Laibowitz pose the next

question: *which applications exist that an average person would find compelling?* [LL05]. Applications developed for sensor networks are still in an incipient or experimental state. Frequently, the actuation scope of the applications is limited to the laboratory where they have been created. Moreover, since there is still no the *killer application*, it seems illusory to envision a massive usage of WSN technology.

3.3. WSN applications profile

In this subsection a snapshot of a typical sensor network application is taken in order to identify the characteristics shared by WSN applications, from the programming point of view. This analysis has been performed independently of the underlying architecture, and it is focused on describing the major features at the application level. The obtained result is shown in the following paragraphs:

- Due to the restricted hardware resources of these devices, the size of the applications developed for WSN should kept as small as possible (around a few hundred code lines). The software is built following a bottom-up strategy. The applications footprint is measured by the RAM and ROM memory consumption[3], which obviously must be lower than the maximum size available in the motes. Developers must, therefore, keep in mind the available hardware budget.

- Applications should spend most of the time in the *sleep* state in order to save energy. Only when a specific event occurs, the application changes its state to *active*, and carries out a small amount of work. When the work finishes, the mote continues sleeping. The energy consumption is a critical aspect of the applications, due to the fact that programming techniques (especially those ones related to the communication strategy) directly affect to the duration of batteries, and, subsequently, the network lifetime.

- Typically, a WSN application is based on local timing, which acts as a reference for performing activities in the mote. This issue is related to the previous paragraph, since the clock events could signal the interchange between different states. In other words, a WSN application could be viewed as a finite state machine directed by events.

- In spite of the wide range of potential WSN applications, it certainly would be possible to identify a basic set of services associated with the physical devices integrated into motes. These services are mostly simple and well-known: read from a sensor, send data, or write into the flash. Note that there is a set of common services for the WSN operating systems.

3.4. Thesis proposal

Taking into account the arguments cited previously, this section establishes the hypothesis and the consequent thesis proposal.

[3]The microcontroller uses physically separated memories for data and code. This approach is known as *Harvard architecture*.

3.4.1. Hypothesis

The hypothesis is stated as follows:

> **Productivity** in wireless sensor networks might be achieved facilitating the applications development through a generic node-centric software architecture, which deals with applications portability in heterogeneous networks and provides enough high-level abstractions to developers, hiding the underlying platform.

According to this hypothesis, this thesis proposes the design, implementation, and evaluation of such generic sensor node software architecture addressing the following challenges:

- Facilitating the WSN applications programming, defining high-level abstractions which can be mapped into the operating system services in a transparent way.
- Generating portable applications across different platforms. One application should be expressed through a uniform design, with independence of the underlying operating system and hardware.

3.4.2. Thesis

While the existing approaches focus on the development of high-level abstractions over a single operating system, our proposal is aimed at masking the underlying hardware and software architectures, providing a common way of writing portable applications. The thesis proposal will consist of the following:

> Design, implementation and evaluation of a sensor node software architecture guiding the development of generic applications which can be ported to heterogeneous platforms, thus increasing the productivity.

The WSN architecture to be proposed should achieve a compromise solution between a reasonable generality while not sacrificing either performance or efficiency. More specific goals of the architecture appear to be the following:

- Separating the concerns about requirements, operation, and specific details such as the MDA standard claims. In other words, the architecture should address a platform-independent development (both operating system and hardware independent). In this way, reusable solutions might be easily interchangeable among different platforms.
- Providing a methodology to specify high-level WSN applications, expressing the relevant information describing a system, and hiding the platform-specific details.
- Covering a reasonable set of platforms (both WSN operating systems and sensor nodes). In other words, the architecture should be multi-platform.
- Proposing a solution at compilation-time (not at run-time), in order to not increase the applications footprint. Applications performance (response time, energy consumption) should not be affected.

3.4.3. Approach and inspiration

Consider an architecture aimed at describing WSN applications with independence of the underlying operating system and hardware. It could be figured out as shown in Figure 3.4. The left side depicts a layered stack where the traditional abstraction levels are present: Hardware (H), operating system (O) and application (App). In order to obtain portability, a Domain Specific Language (DSL) could be described, whose goal would be to specify the operating systems abstractions and constructs allowing sensor node-centric programming to be expressed. Subsequently, applications are written using such DSL. An additional layer, the *Operating System Abstraction Layer* (OSAL), will be incorporated. Its goal is to translate generic applications written using this DSL into the equivalent operating system-specific implementation. As examples of such architecture, different components at each abstraction level are presented on the right side. An architecture could be composed of the combination of one component belonging to every layer.

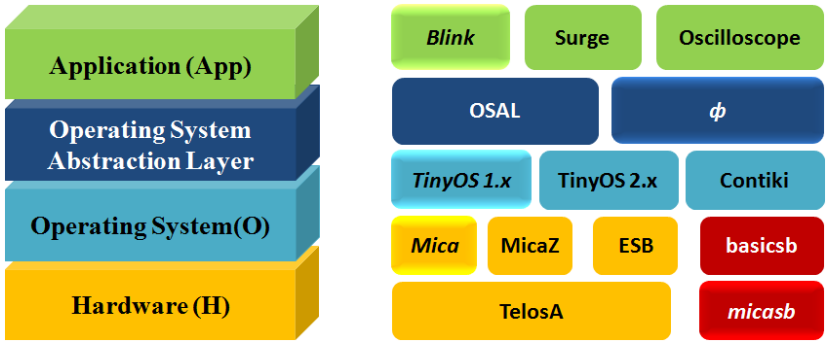

Figure 3.4: Software architecture of a sensor node. On the left side, a layered stack identifying the traditional abstraction levels. On the right side, a set of components at every level. Note that the components *Blink, ∅, TinyOS 1.x, {Mica, micasb}* have been highlighted to indicate a possible instance of architecture.

As previously mentioned, WSN applications are currently platform-dependent, thus, sensor network architectures lack a translation layer to isolate the applications from the low-level details. In terms of MDA standard (refer Section 2.5), the current applications would map a Platform Specific Models (PSMs) of a system[4]. Abstractions on top of the current architecture to increase the portability and ease the programming are described in this thesis work. Analogously, these abstractions must map Platform Independent Models (PIMs). According to MDA, a *transformation process* must be defined to translate PIMs to PSMs. Figure 3.5 compares the current applications building model, in which only PSMs are obtained, to the proposed model, which incorporates notations to define systems in a higher abstraction level (PIMs).

Following this approach, this thesis proposes using the MDA standard to describe the architecture dealing with the applications development. It implies specifying both the formalisms to describe platform-independent and generic applications through a *Domain Specific Language* (DSL), and the transformation method into the WSN operating system-specific code. Subsequently, a translator between operating system and application (such as Figure 3.4 depicts) will be implemented. This layer is called the *Open Services Abstraction Layer* (OSAL), and will be

[4]We will assume that applications requirements have been previously collected, and, in this way, the Computation Independent Model (CIM) was elaborated.

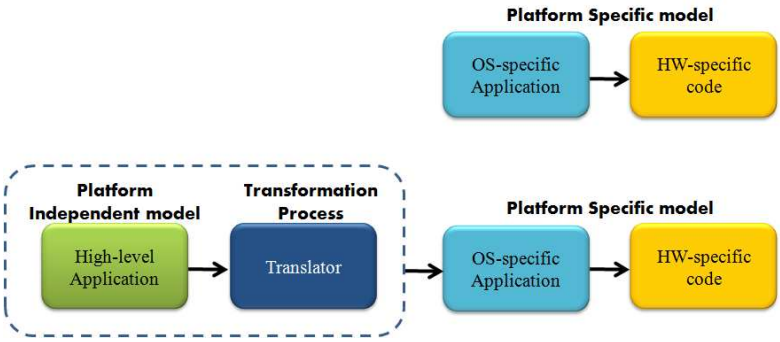

Figure 3.5: A generic MDA process: from Platform Independent Model (PIM) to Platform Specific Model (PSM).

explored in further chapters. Figure 3.6 presents the three steps in the code generation process.

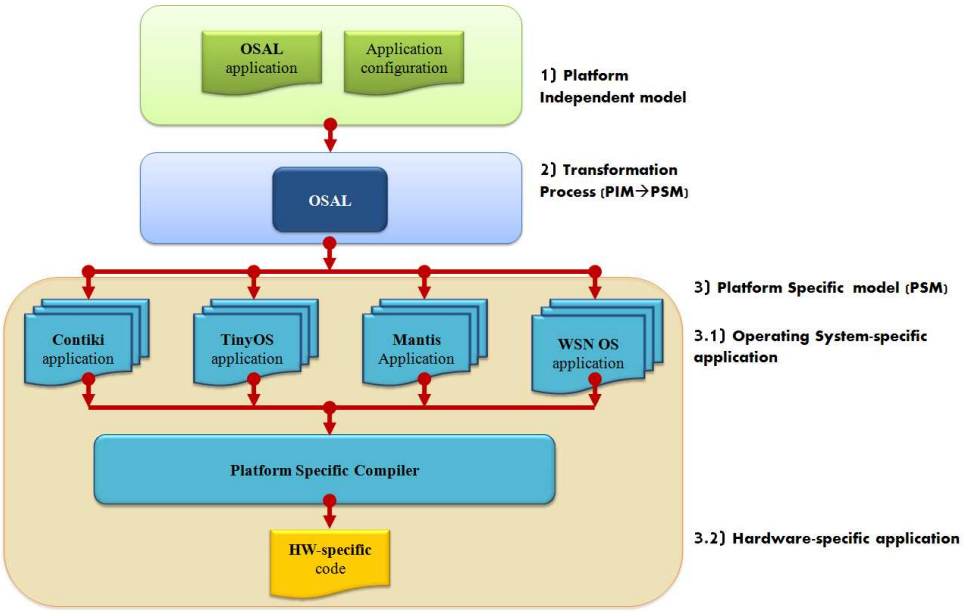

Figure 3.6: MDA standard and WSN applications programming.

The three steps can be summarized as follows:

- The Platform Independent Models (PIMs) description will consist of the formal definition of portable applications on top of the WSN operating system using a DSL. DSL offers a standardized interface of programming and a set of declarative sentences to express the behavior of sensor nodes (node-centric programming approach). Therefore, DSL applications must be written using these abstractions.

- The transformation process will consist of the implementation of different compilers of the Domain Specific Language into the equivalent operating system-specific code, in an automatic or semi-automatic way.

- The Platform Specific Models (PSMs) description will be generated from the previous stage, and represent the operating system-specific description of the high-level application (DSL application).

3.5. Chapter summary

In this chapter some important obstacles in the wireless sensor networks development such as applications portability and development process have been identified. The relevance of the problem lies in the complexity of the programming, making it difficult to extend the technology to the non-expert developers community, and, consequently, limiting the usability of technology.

Firstly, we have analyzed the problem focusing on the hardware and operating system level. Secondly, we have considered every operating system execution model in order to determine how it affects the programming model. Thirdly, some evidence of the lack of portability has been shown for different operating systems.

Once the problem has been stated, the hypothesis of this work proposes favoring the productivity of WSN applications by the proposal of a new architecture, which is focused on easing a applications building and increasing portability. Subsequently, this thesis plans to design, implement, and evaluate a software sensor node-centric architecture dealing with these relevant challenges. An overview and the major objectives are also identified. The solution proposed is inspired by the MDA standard, which claims to separate the independent and specific concerns of a system.

This description marks the starting point for further chapters.

Chapter 4

Sensor node-centric architecture

This chapter introduces the architecture proposed to satisfy the hypothesis formulated in the previous pages. Accomplishing the task of design implies reviewing the original architecture: the deconstruction of a *generic* sensor node will be carried out. *Generic* means that the sensor node abstracts away specific details, focusing on functionality, interface, and structure. The sensor node-centric architecture is designed in a multi-layered fashion, where the responsibilities of each layer are clearly defined. Every layer is modeled using UML 2.0 diagrams, allowing its description with no ambiguity. The proposed architecture is susceptible to being formalized through the theory of sets. In this way, the fundamentals of the architecture are clearly established.

The chapter is also intended to describe the platform-dependent design, focusing on the instantiating of different components. HW is modeled and OS is analyzed like a black box connecting applications and hardware: it exports a services interface (input) and transforms it into hardware requests (output). Hardware components are also described using XML manifests following a predefined scheme. Each OS provides an heterogeneous interface, very different both in semantics and in syntax. Once those prerequisites have been established, the next step is to present the overlying architecture, which represents the most remarkable contribution of the thesis work. The upper layers over the OS are intended to homogenize the access interface to write platform-independent applications. Their study is carried out in the following chapters.

4.1. Fundamentals of the architecture

As mentioned, the proposed architecture finds its main motivation in the lack of approaches addressing the portability and application development challenges for WSN programming. With this goal in mind, the following design principles and fundamentals have been considered:

- *Sensor node-centric*. The architecture is focused on studying a generic sensor node as a whole and not as a part of the network (therefore following the node-centric approach). A generic sensor node is a self-contained device presenting a set of common features at different abstraction levels.
- *Multi-platform*. Masking the specific details in a platform-independent design implies that the architecture should be HW and OS independent. Additionally, the architecture implementation should consider a reasonably broad set of platforms. The hardware abstractions are provided by the OS, while the overlaying architecture should deal with the OS abstraction.

- *Multi-layer* design. Every sensor node abstraction level is modeled as a component or *layer*, which encapsulates a certain functionality. Every layer can interact with the immediate upper and lower one through a predefined interface, which is provided and used respectively. It means that every layer is itself a connector between both interfaces. Additionally, to ensure an independent architecture, layers should be interchangeable among different instances of specific sensor nodes.
- *Programming abstractions*. Applications must be built on top of the architecture independent of the underlying platform, and a compiler over the OS layer should deal with the complexity of the translations between both abstraction levels. Thus, such programming abstraction allows *generic* WSN applications to be written.
- *Generation of code*. This process takes a *generic* application (native application) as input and transparently transforms it into the equivalent platform-specific application (final application). In this way, the tailored code is automatically generated, which can be also compiled for a particular sensor node to obtain the executable code. The same single *generic* application could be translated to different platforms.
- *Reusability*. The proposed architecture should be able to reuse the basis of the original architecture when possible: in particular, existing hardware and operating systems should be taken into account. Moreover, every component in the sensor node architecture could be reused, and the integration should be supported.
- *Minimal overhead*. The overhead imposed by the architecture on the target applications should be as minimal as possible in order to respect the limited hardware budget. This means that the translation relation between the high-level application and the operating system-specific one must be so exact as possible.

The requirements of separation of concerns, generic models, and subsequent platform-dependent implementations become the architecture proposed in a MDA-based architecture. In this way, a specific sensor node can be viewed as a particular instance of a *generic* sensor node. To obtain high-level abstractions, a *Domain Specific Language* (DSL) intended to write generic and operating-system independent applications will be specified. In the underlying level, an *Operating System Abstraction Layer* will be incorporated, which is intended to carry out transformations of DSL applications built on top of the architecture into OS-specific applications. Therefore, this layer can be viewed as an *applications generator*. Note that these translations are not limited to API conversions simply, but a complete process of adaptation is required, which is inherent to the WSN operating-system. The output will be the tailored code to deploy in the target platform. As shown in the next chapters, the overhead imposed is minimum. The advantages of this approach can be summarized as follows:

- Ease of applications programming by a high-level interface friendly to applications.
- Applications portability, due to the applications developed can be transparently translated to the underlying platform with no additional work.
- Clear and platform-independent applications design, incorporating the required knowledge while specific details are kept hidden.
- Flexibility to incorporate new components and ease of integration. Components are interchangeable with other analogous ones in the architecture.

4.1.1. Scope and restrictions

As described, the architecture provides a development framework for WSN applications programmers. However, it is necessary to establish the scope: the set of applications and platforms for which it is considered valid. There are two boundaries found:

- The set of potential applications, which can be developed using the proposed architecture are determined by the set of abstractions provided by the DSL, especially, by the supported services. If the DSL does not offer a certain functionality required by one application, such application cannot be expressed. Therefore, the richer the interface, the bigger the set of applications that can be produced.
- The set of hardware platforms supported by the proposed architecture is given by the underlying OS. *Ad-hoc* transformations will be done to a selected OS, which determines the set of available platforms. These available platforms are those for which the OS has been ported. In other words, the architecture does not provide hardware portability at the OS level, and only those platforms currently supported by the OS will be considered.

4.2. Architecture design

This section describes the sensor node-centric architecture proposed. As previously mentioned, it has been designed in a multi-layered approach, integrating both the traditional WSN architecture, basically composed of hardware and operating system, and eventually of a intermediate Hardware Abstraction Layer (HAL), and two upper layers: an *Operating System Abstraction Layer* and *Application Layer*. The architecture is represented in Figure 4.1.

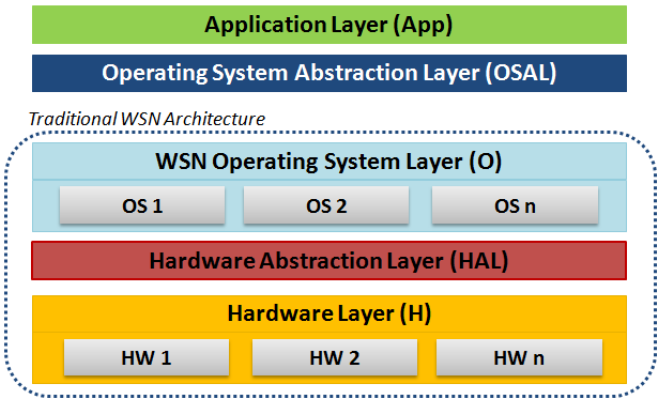

Figure 4.1: Sensor node-centric architecture design.

The goal of the two upper layers is to offer a set of portable and open services hiding the platform. The *Application Layer* is intended to provide the required abstractions and establish the writing rules to program generic applications on top of the traditional WSN architecture. For it, a complete, generic, and robust DSL will be specified. The goal of *Operating System Abstraction Layer* is to translate the DSL application into the functionally equivalent one written using the underlying WSN operating system. *Operating System Abstraction Layer* and *Application Layer* are extensively described in subsequent chapters, while the current chapter accomplishes their integration with the underlying levels: *Hardware* and *Operating System*. The same idea can be also

represented using an *implementation diagram* of UML 2.0 (particularly, component diagrams), as in Figure 4.2. However, some additional information is provided thanks to these kinds of diagrams. The architecture is split in the mentioned layers. Every layer is encapsulated into a component. Every component exports a predefined interface to the upper layer and uses the interface offered by the immediate lower level. Such a description is made from a platform-independent point of view, and, any sensor node can match this template. The figure represents a generic architecture divided at different layers: every component can be instantiated to compose a specific architecture. Note that the traditional architecture is *monolithic*. The only restriction is that the component must respect the interaction with the immediate upper and lower layers through predefined interfaces. In this way, instances of the OS layer could be TinyOS 1.x or Contiki, and by extension, any WSN operating system exporting the proper interface. Analogously, Mica, Telos or ESB sensor nodes could map the hardware layer. Subsequent sections study in depth each one of these levels.

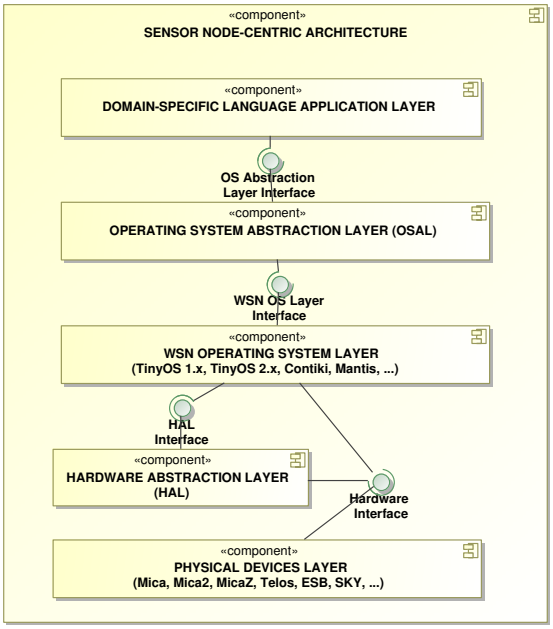

Figure 4.2: Implementation diagram of sensor node-centric architecture.

4.2.1. Hardware Layer

Figure 4.3 depicts a UML implementation diagram representing the *Hardware Layer* of the architecture (H). It includes the physical devices held in sensor nodes, and the main settings identified. Every component exports a specific interface, related to the functionality associated with the physical device. Note that such interfaces provide the lowest-level services in the architecture. Conceptually, the whole set of services provided by the hardware is composed of the union of the sub-interfaces due to the physical components, which can be expressed in the following way:

$$HI = HI_{microcontroller} \cup HI_{sensorboard} \cup HI_{radio} \cup HI_{bus} \cup HI_{flash} \cup HI_{leds}$$

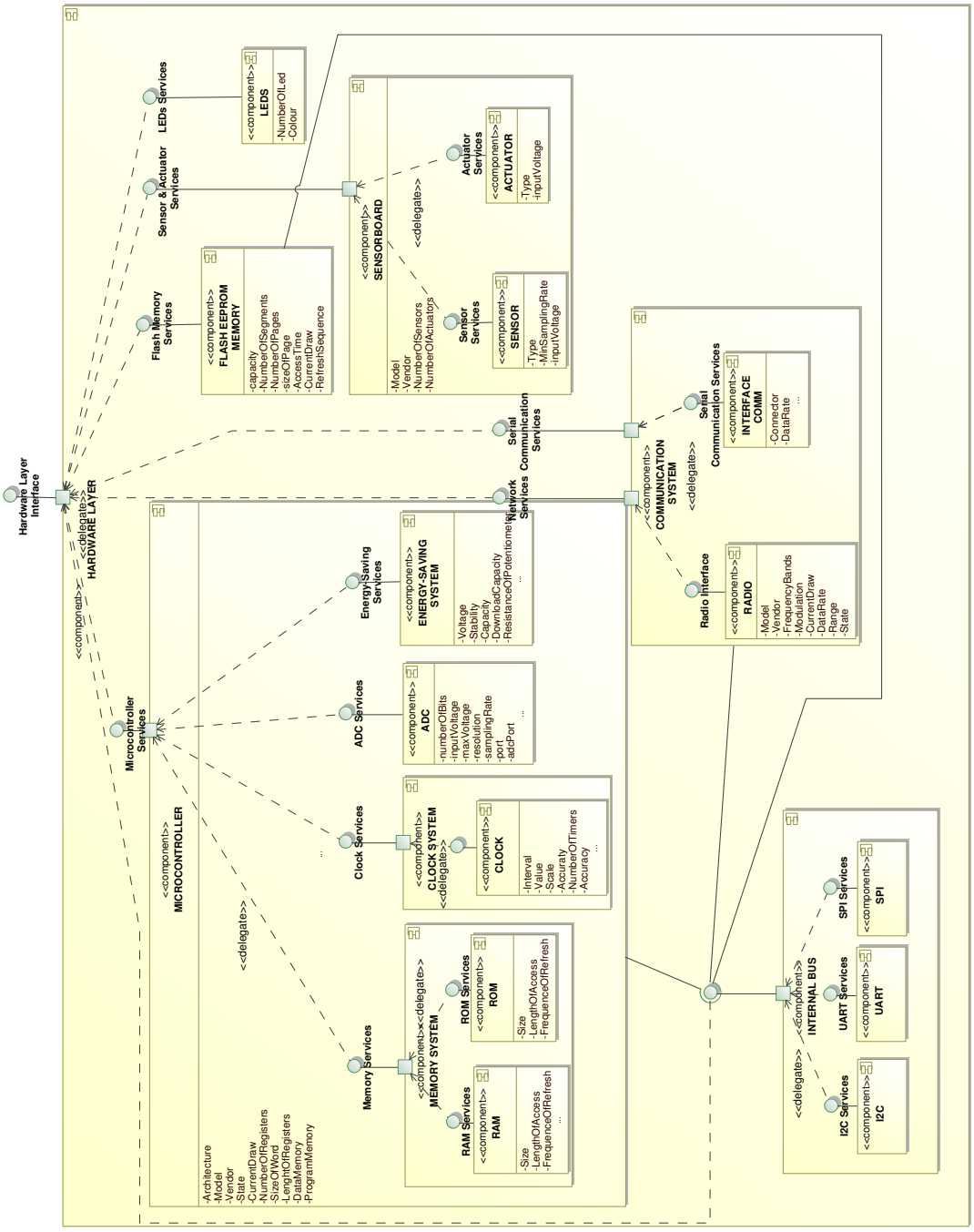

Figure 4.3: Implementation diagram of the *Hardware Layer*.

4.2.1.1. XML Schema and Manifest

In order to describe the components of a sensor node, a powerful and flexible specification language is required. *XML Schemas* are proposed for components description. One XML Schema must be created for each type of component (e.g. CPU, radio, sensor board or sensor) in order to completely characterize its features, operations and restrictions. In this way, XML Schemas will act like a template to specify components of the same type.

Every instance of component type (e.g. radio CC2400 of type `Radio`) is described through a *XML Manifest*, i.e. an XML [CHA08] file which contains the characteristics, interfaces and compatibilities of a particular hardware. XML Manifests must be manually created by the programmer in accordance to its XML Schema (or DTD). In this way, XML Schemas act as pattern of devices of the same kind. Therefore, every component should be described by one XML Manifest, which must be in accordance with its XML Schema.

XML Schemas and Manifests have been chosen because they allow to decouple the definition from the instantiation of a component, and to make flexible, adaptable, and rigorous definition of components.

The following data is considered relevant to describe a component:

- *Signature*. The signature is a unique identifier to distinguish each instance of an XML Manifest file, i.e. for every component. The signature could be a concatenation of the model and the serial number of the component because the serial number is always unique for each product of a model.
- *Definitions*. This section completely identifies the properties of the component. It contains information such as the model, the supplier, and other specific settings related to the device to be described.
- *Assemblies*. In this section a relation of executable files associated with the device is listed, for example, the code provided by the supplier (e.g. device drivers).
- *Interface*. Every component is a software piece that can interact with another through interfaces. The interfaces define both the behavior required or imported and the behavior provided or exported by the component. Thus, it is necessary to distinguish between the interfaces *Import* and *Export*, and the operations that the component requires and provides, respectively.
- *Resources*. The resources that one component is allowed to access. In this way, the set of available resources for one component can be restricted. The way to identify resources could be the signature of the component and the identifier of the resource.
- *Compatibility* defines the architectural components that can be accessed from the component that we are defining. In other words, a component is allowed to interact with another one if and only if this fact has been previously enumerated in the *Compatibility* section.

4.2.2. Operating System Layer

Figure 4.4 shows the design of the *Operating System Layer* (O). This layer is composed of several components encapsulating the functionalities derived from the physical hardware devices. The interface of this layer (in the Figure named *Operating System Interface*) is the union of all

the interfaces of every functionality integrated at the OS component. The OS interface could be expressed as following:

$$OI = OI_{scheduling} \cup OI_{sensors\&actuators} \cup OI_{communication} \cup OI_{storing} \cup OI_{debugging} \cup OI_{clock\&energy\ \ saving}$$

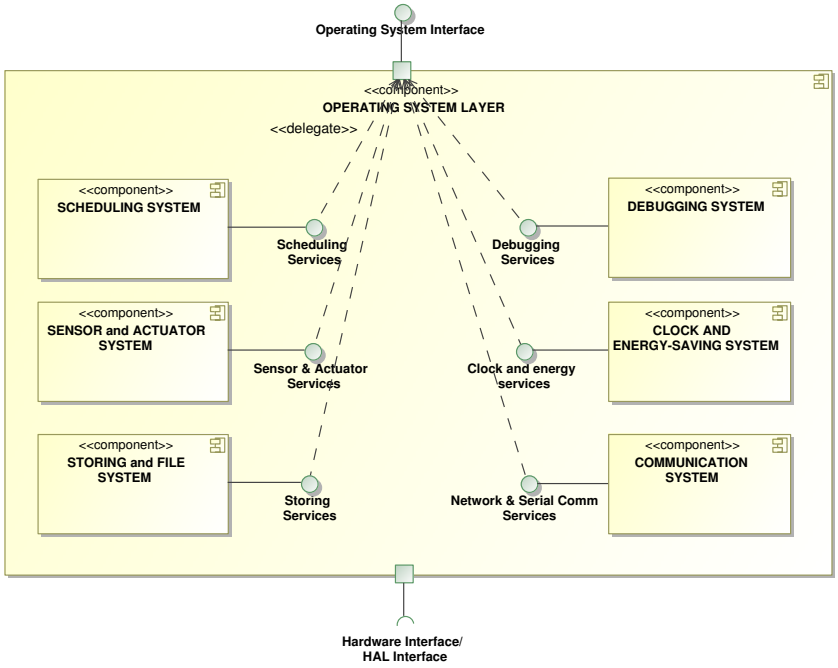

Figure 4.4: Implementation diagram of *Operating System Layer*.

This interface is provided to the upper layer, in the proposed architecture, the *Operating System Abstraction Layer*. Analogously, OS uses the HW (or eventually HAL) interface. In the following sections of this chapter, for each functionality the interface is identified for a set of operating systems chosen.

4.2.3. Operating System Abstraction Layer

The upper layers located on top of the traditional WSN architecture previously defined are intended to perform the effort of masking the underlying platform. These layers are the *Operating System Abstraction Layer* and *Application Layer*, which have been created from scratch to address the problem stated in Chapter 3, and both constitute the major contribution of the thesis work.

In the architecture proposed, the *Operating System Abstraction Layer* is an intermediary layer with the aim of abstracting away the WSN operating system. It has as its objective to translate the written application using a *Domain-Specific Language* into the WSN operating system-specific implementations (e.g. TinyOS, Contiki), which can be then translated into binary code by the target compiler. In order to perform this translation, both the high-level language and its compiler must be defined.

For the first one, a *Domain Specific Language* (DSL) with the purpose of describing a textual notation for portable applications among heterogeneous platforms must been elaborated. Therefore, it must offer the set of constructs and functionality, which can be obtained from the analysis

of different WSN operating systems interfaces. Subsequently, the OSAL interface (OSALI) should encapsulate the specific OSes services, subsequently:

$$OSALI = OSALI_{scheduling} \cup OSALI_{sensors\&actuators} \cup OSALI_{communication} \cup OSALI_{storing} \cup OSALI_{debugging} \cup OSALI_{clock\&energy\ \ saving}$$

Note that in order to obtain portability, which is the main goal of this thesis, the grain of these operations (or abstraction level) could be the same as that offered by the operating system, while the provided interface (OSALI) must be unique for any underlying operating system. In addition to the above, every high-level primitive can map one or several low-level primitives.

For the second one, it is necessary to define the proper compiler which translates the OSALI primitives to the corresponding low-level calls in every particular OS. While the exported interface must be homogeneous, primitives should map the synonymous services in each OS. In other words, the relation between a high-level service (from OSALI) and its corresponding OS service (from OI), is basically semantic. Therefore, the *Operating System Abstraction Layer* is intended to perform the services translation between two different abstraction levels: application and OS.

Figure 4.5 proposes a possible structure for *Operating System Abstraction Layer*. As shown, it is composed of three main components: *interface*, *pre-compiler* and *translator*. The first one must provide a complete programming language to write generic applications independently of the underlying architecture. The second one is intended to verify the syntax of the high-level application, according to the writing rules established. Finally, the translator has as its goal to parse the code (native application) and generate the equivalent code (target application), which will use the OS interface.

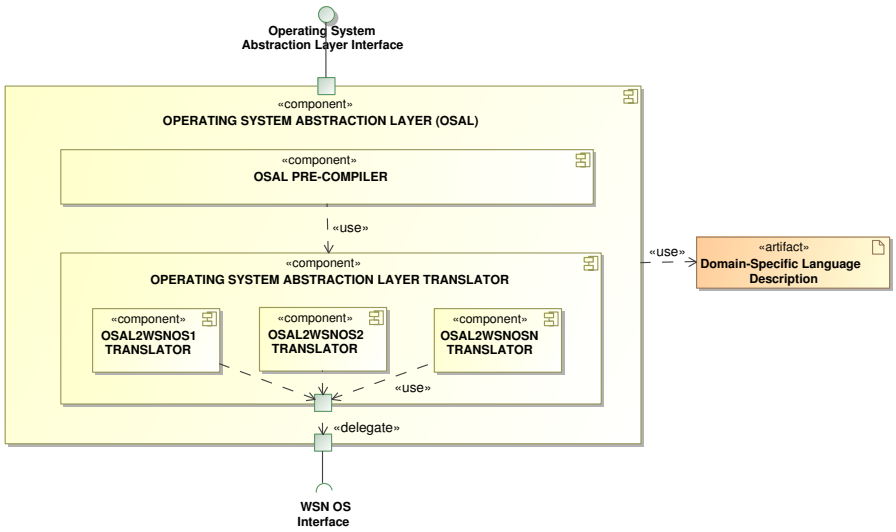

Figure 4.5: Implementation diagram of *Operating System Abstraction Layer*.

The main advantage is the clear separation between the different development levels, also favoring the division of responsibilities: programmers could focus on the aspects related to applications without taking into account the complexity inherent to each platform.

4.2.4. Application Layer

The highest layer has been referenced as the *Application Layer*. It represents the complete domain of WSN applications that are able to interact with the proposed sensor node-centric architecture. The applications programming must respect the notation provided by the DSL described and interact with the underlying level, the *Operating System Abstraction Layer*.

4.3. Architecture formalization

The design previously described is susceptible to being formalized, due to that both scope and responsibility in each layer has been clearly established. Theory of sets is used to formally describe the sensor node-centric architecture. It is necessary to provide an explanation of how to assemble a WSN architecture from a set of well-known components or layers, and to represent the set of potential instances of architecture. It is assumed that such architecture is composed of four layers: hardware, OS, OSAL and application, which are formalized as follows:

Let A be the set of WSN software architectures. While A has been traditionally monolithic, let A be now defined as the Cartesian product of the set of WSN hardware platforms, operating systems, the abstraction layers described over them and applications:

$$\mathbf{A} = \mathbf{APP} \times \mathbf{OSAL} \times \mathbf{O} \times \mathbf{H} = \{(a, b, c, d) \mid a \in \mathbf{APP} \wedge b \in \mathbf{OSAL} \wedge c \in \mathbf{O} \wedge d \in \mathbf{H}\} \quad (4.1)$$

where *APP* represents the potential set of WSN applications:

$$\mathbf{APP} = \{Blink, Surge, BaseStation, \ldots\} \text{ (By extensional)} \quad (4.2)$$

OSAL corresponds to the set of abstractions defined on the WSN operating system:

$$\mathbf{OSAL} = \{SN\text{-}OSAL, \ldots, \varnothing\} \text{ (By extensional)} \quad (4.3)$$

O is the set of operating systems specifically designed for sensor nodes:

$$\mathbf{O} = \bigcup O_i, \forall i = 1, ..., n \quad (4.4)$$

and H is the set of sensor nodes defined as the Cartesian product between platforms (specifically motes) and sensor boards:

$$\begin{aligned} \mathbf{H} = P \times S &= \{(a, b) \mid a \in P \wedge b \in S\} \\ P &= \{\text{mica, mica2, telos, telosb, sky, esb, } \ldots\} \text{ (By extensional)} \\ S &= \{\text{mts310, mts300, mda300, } \ldots, \varnothing\} \text{ (By extensional)} \end{aligned} \quad (4.5)$$

From an architectural point of view, each one of these sets (except the upper one) can be described as the union of two subsets: functionality (F) and interface (I). Thus:

$$\begin{aligned} H_i &= \{HF_i \cup HI_i\}, \forall H_i \in H \\ O_j &= \{OF_j \cup OI_j\}, \forall O_j \in O \\ OSAL_k &= \{OSALF_k \cup OSALI_k\}, \forall OSAL_k \in OSAL \end{aligned} \quad (4.6)$$

Note that the interface of the k Operating System Abstraction Layer is denoted as $OSALI_k$, and it corresponds to the interface provided by the high-level DSL for writing high-level applications.

Let $\mathcal{R}_T$ be defined as the translation relation over the Cartesian product of OSAL × O. Then, $\mathcal{R}_T \subseteq OSAL \times O$, such that:

$$\begin{aligned} OSAL \times O &= \{(OSAL_k, O_j) \mid OSAL_k \in OSAL \wedge O_j \in O\} \\ \mathcal{R}_T &= \{(a,b) \in OSAL \times O \mid \\ \forall x \in OSALI_a\ \exists y \in OI_b, &\ OSALI_{a,x} \textit{ is semantically equivalent } OI_{b,y}\} \\ OSAL_k \mathcal{R}_T O_j &\iff (OSAL_k, O_j) \in \mathcal{R}_T \end{aligned} \tag{4.7}$$

Note that y represents a functionality and not a single primitive, and therefore, it could map more than one basic service.

Given such translation relation between the sets $OSALI$ and OI, it is possible to define $\succ_T$ as the function to compute the mapping between OSAL and the operating system interface. Then, it is denoted as $\succ_T$: $OSALI_k \rightarrow OI_j$, and it satisfies:

$$\succ_T (x) = y, \forall x \in OSALI_k,\ \exists y \in OI_j, OSAL_k \in OSAL \wedge O_j \in O \mid OSAL_k \mathcal{R}_T O_j \tag{4.8}$$

where $OSALI_k$ is the interface of $OSAL_k$ (as previously described, the DSL interface), which is exported to the application level, and uses the OI_j OS interface. The $OSALI_k$ should cover the functionality offered by each $O_j \in O$. Therefore, $\succ_T$ is a function which finds its maximum and minimum representation at the following limits:

$$\begin{aligned} max(\succ_T(OSALI_k)) &= \bigcup OI_j\ \forall O_j \in O \\ min(\succ_T(OSALI_k)) &= \bigcap OI_j\ \forall O_j \in O \end{aligned} \tag{4.9}$$

The objective for this interface is:

$$max \mid (OSAL_k, O_j) \in \mathcal{R}_T \mid \forall O_j \in O \wedge \text{k is unique} \tag{4.10}$$

At the underlying level, let $\mathcal{R}_P$ be a portability relation defined over the Cartesian product of O × H. Then, $\mathcal{R}_P \subseteq O \times H$ such that:

$$\begin{aligned} O \times H &= \{(O_j, H_i) \mid O_j \in O \wedge H_i \in H\} \\ \mathcal{R}_P = \{(a,b) \in O \times H \mid \forall x \in OI_a\ \exists y \in HI_b, &\ OI_{a,x} \textit{ is semantically equivalent } HI_{b,y}\} \\ O_j \mathcal{R}_P H_i &\iff (O_j, H_i) \in \mathcal{R}_P \end{aligned} \tag{4.11}$$

Subsequently, let $\succ_P$ be the function defined to compute the mapping of the operating system interface into hardware interface. Then, it is denoted as $\succ_P$: $OI_j \rightarrow HI_i$, and it satisfies:

$$\succ_P (x) = y, \forall x \in OI_j,\ \exists y \in HI_i, O_j \in O \wedge H_i \in H \mid O_j \mathcal{R}_P H_i \tag{4.12}$$

Given this formalization, the subset of WSN valid architectures is a subset of A satisfying:

$$A' = \{(a,b,c,d) \mid a \in APP \wedge b \in OSAL \wedge c \in O \wedge d \in H, (b,d) \in \mathcal{R}_T \circ \mathcal{R}_P\} \tag{4.13}$$

where $\mathcal{R}_T \circ \mathcal{R}_P$ is a composite relation such that:

$$\mathcal{R}_T \circ \mathcal{R}_P = \{(OSAL_k, H_i) \mid OSAL_k \in OSAL \wedge H_i \in H,\ \exists\, O_j \in O, (OSAL_k, O_j) \in \mathcal{R}_T \wedge (O_j, H_i) \in \mathcal{R}_P\} \tag{4.14}$$

Algorithm 4.1 presents the process of building mapping rules between $OSAL_k$ and a specific operating system O_j. Such a function has been denominated $\diagdown\!\!\!\!\succ_T : OSALI_k \rightarrow OI_j$. As shown, for each primitive in the OSAL interface, a semantically equivalent primitive in the OS interface is treated of finding. If it does not, then such a primitive is void, and it should be implemented by $OSAL_k$.

Algorithm 4.1 Generic process for mapping OSAL into OS interface $\diagdown\!\!\!\!\succ_T : OSALI_k \rightarrow OI_j$

```
for all x in OSALI_k do
  for all y in OI_j do
    if x is semantically equivalent y then
      ⋉_T(OSALI_{k,x}) = OI_{j,y}
  if (!mapping found(x)) then
    ⋉_T(OSALI_{k,x}) = ∅
```

Let APP be the set of valid applications built on top of the architecture A_n. Then APP_l is an application that satisfies that:

$$APP_l \in APP$$
$$A_n = \{(APP_l, OSAL_k, O_j, H_i) \mid APP_l \in APP \wedge OSAL_k \in OSAL \wedge O_j \in O \wedge H_i \in H, (OSAL_k, H_i) \in \mathcal{R}_T \circ \mathcal{R}_P\}$$
$$APP_l = \bigcup x,\ \mathrm{x}\ \in \{DSL_k, \varnothing\} \tag{4.15}$$

where DSL_k represents the DSL to be compiled by the k OSAL. It must include the set of instructions to construct programs on top of the OS according to the writing rules agreed, and the interface $OSALI_k$. Therefore, the set of OSes for which the APP_l application could be automatically generated by $OSAL_k$ is represented as:

$$APP_l \textit{ can be automatically translated } \forall O_j \in O \Longleftrightarrow OSAL_k \mathcal{R}_T O_j \tag{4.16}$$

4.3.1. Analysis of functions

This section analyzes the properties of the functions used in the previous formalization to state with no ambiguity the relation between the elements of domain and codomain sets.

4.3.1.1. Properties of $\diagdown\!\!\!\!\succ_T : OSALI_k \rightarrow OI_j$

$\diagdown\!\!\!\!\succ_T : OSALI_k \rightarrow OI_j$ is a mathematical function to establish a relation between the API of the predefined high-level DSL for the k Operating System Abstraction Layer ($OSAL_k$), in the API of every underlying WSN OS (OI_j) (or codomain set). Its properties are:

- $\diagdown\!\!\!\!\succ_T$ must be an *injective* function given that it satisfies:

$$\forall x,y \in OSALI_k,\ \diagdown\!\!\!\!\succ_T (\mathrm{x}) = \diagdown\!\!\!\!\succ_T (\mathrm{y}) \Longleftrightarrow x = y \tag{4.17}$$

It is important to point out that elements of both sets are functionalities instead of single primitives (for instance, read a sensor could imply several primitives). In this function there could exist non-selected elements from the codomain set, while each element from $OSALI_k$ set must match another one from the OI_j set. Note that this is a design requirement, and it is considered *mandatory*.

- $\rangle_T$ could be also *surjective* function such that:

$$\forall y \in OI_j, \exists x \in OSALI_k \text{ such that } y = \rangle_T(x) \tag{4.18}$$

The fulfillment of this property ensures that all functionalities in every target WSN operating system are the image of one described high-level functionality. However, the challenges found in becoming the $\rangle_T$ function in a surjective function are:

- Heterogeneous interfaces offered by the current WSN OSes, and not in agreement with on the functionality provided.
- Rapid growing, new OSes and new updates of the OSes.
- Lack of standardization of both interfaces and functionality.

Subsequently, although covering the total spectrum of services provided by all WSN operating systems should be the optimal condition for the architecture, note that this requirement is *optional*, and it does not prevent correct behavior, but rather limits the set of potential applications to be expressed. In the case of becoming such condition, $\rangle_T$ function should also be bijective.

4.3.1.2. Properties of $\rangle_P : OI_j \rightarrow HI_i$

$\rangle_P : OI_j \rightarrow HI_i$ is a mathematical function intended to establish a relation between the API of a WSN operating system in the API of every supported hardware platform. Its properties are:

- $\rangle_P$ is a *bijective* function, given that only the set of hardware platforms that have been ported to a specific operating system *j* are considered. It is also assumed that all functionality expressed at the hardware level finds a related bigger grained operation at the upper layer.

4.4. Architectural models

Once the design principles and formalization are presented, it is time to go into details about the part of the architecture taken as heritage: HW and OS. In this way, the traditional architecture could be reused and integrated into the design proposed. This section is focused on describing both levels through specific instances, and subsequently, their components, functionality, and overall interface have been identified. The overlying architecture description was also accomplished. In particular, the *Operating System Abstraction Layer* has been also instantiated, whose analysis is carried out in the following chapters, while a preliminary overview is given in this chapter.

4.4.1. Hardware Layer instantiation

The major difficulty found in modeling the *Hardware Layer* is the heterogeneity of the physical devices integrated into it. This section focuses on identifying the main components at the

hardware level, settings and functionality. An interface has also been determined for each device. However, as mentioned, the proposed architecture does not offer hardware portability at the OS level, but it just assumes the hardware platforms currently supported by every OS. In other words, the function of portability between a pair $\{HW_i, OS_j\}$, denoted as $\succ_P$ in the architecture formalization, will only be possible if the OS supports that HW, and no additional effort have been accomplished at this scope. It is important to point out, that the OS is responsible for mapping high-level abstractions into the HW abstractions, which would stay hidden for the levels on top of the OS.

4.4.1.1. Microcontroller

The microcontroller is the core of the sensor node, and its management used to imply operating over the devices, such as one or more clocks, Analog-to-Digital converter (ADC) and RAM and ROM memories. Additionally, depending on the microcontroller, it could offer mechanisms to gradually manipulate its power state (which ranges from *idle* to *active* mode). As a general rule, the microcontroller should always be in the lowest energy level that satisfies the application requirements. The most simple microcontrollers consume constant power to execute instructions [SHrC+04], while in more sophisticated microprocessors, power management at chip level is incorporated. On the other hand, in some cases it is the OS itself which does not give support to that feature. Some examples of microcontrollers were described in Chapter 2. Subsequently, the operations arising from the microcontroller are related to the clock system management, ADC operations, memory management and power saving management. Table 4.1 expresses in an informal and general way the interface covering such functionality.

Device	Operation
RAM memory	readMemory(int *dir, char *buf, int length) writeMemory(int *dir, char *buf, int length)
ROM memory	readByte(int *dir, char *byte) writeByte(int *dir, char *byte)
Clock system	startTimer(int timer, int interval) stopTimer(int timer) getCounter() setCounter(int value)
Energy saving system	adjustPower() setMode(int mode) int getMode()
ADC	initialize() analogicalToDigital(int channel, int port)

Table 4.1: An example of microcontroller interface.

4.4.1.2. Radio

The radio device is intended to give network communication capacity to sensor nodes. Characterizing radio devices is a complex task because it requires the integration of very heterogeneous devices into a single model. There is not a unique communication standard, with different possibilities existing: from Bluetooth standard (e.g. Intel mote or BTnode) to ZigBee (e.g. MicaZ mote), which even could use other frequency bands not compatible with the previous ones (e.g.

Mica and Mica2 motes). The diversity of features and settings is shown in Table 4.2 for some popular radios held in sensor nodes, which were shown in Chapter 2.

Radio	TR1000	CC1000	CC2400	nRF2401	CC2420[1]
Manufacturer	TRM	Chipcon	Chipcon	Nordic	Chipcon
Frequency Band	868/915 MHz	300/1000 MHz	2.4Ghz	2.4Ghz	2.4Ghz
Max. Data Rate (kbps)	115.2	76.8	1000	1000	250
RX power (mA)	3.8	9.6	24	18(25)	19.7
TX power (mA/dBm)	12 / 1.5	16.5 / 10	19 / 0	13/0	17.4/0
Powerdown power (μA)	1	1	1.5	0.4	1
Turn on time (ms)	0.02	2	1.13	3	0.58
Modulation	OOK/ASK	FSK	FSK/GFSK	GFSK	DSSS/O-QPSK
Packet detection	No	No	Programmable	Yes	Yes

Table 4.2: Radio settings.

The radio device management could involve the development of protocol stack to provide sensor nodes the transmission and reception of data to and from WSN. Operations associated with radio devices are related to transmission and reception data through the physical medium, to power on and off the device, to switch between transmission and reception mode, and other additional functions depending on the chip.

4.4.1.3. Sensor boards, sensors, and actuators

Currently, there is a wide variety of available sensors and actuators to be integrated into sensor nodes, as was shown in Chapter 2. Depending on the application requirements, a sensor node could accommodate several sensors. In some cases sensors are placed on the board itself (e.g. Telos family motes) or incorporated into a sensor board, which hosts several sensors and actuators.

In conjunction with the ADC device, the main challenge is to be able to efficiently sample different sensors to obtain analogical data which can be transformed into digital values. It is carried out in the following way: the microcontroller throws a read request to a specific sensor (note that one single measurement can be taken each time). When such a sample has been captured, it is transformed into a digital value through the ADC. Then, the microcontroller is interrupted, and the digital value is delivered using the internal bus (typically a I2C or SPI bus).

4.4.1.4. Flash EEPROM memory

Flash memory chips integrated into sensor nodes are intended to permanently store data arriving from different sources: data collected from sensors, configuration data and binary program images. Flash devices are *Electrically-Erasable Programmable Read-Only Memory* (EEPROM) memories allowing several memory positions to be written or erased in a single programming operation. Flash memory chips are of two types: NOR or NAND. In NOR chips, the flash cell is in its default state, which means the equivalent of a binary ”1” value. Every NOR flash cell can be programmed, or set to a binary ”0” value. The NAND technology works in the inverse way. The erase operation restores the original state. Flash memory chips also present other important features:

[1] ZigBee Compliant.

- Pages can only be written or erased as a whole. Pages should be erased before being written in order to maintain consistency.
- Flash memory pages can only be rewritten a limited number of times, typically about 10.000 operations. This problem is commonly known as *wear leveling*.

Examples of popular flash memory chips integrated into sensor nodes were described in Chapter 2. Table 4.3 depicts some technical settings of different flash memory chips. The main usage of the flash memory chip is the building and management of file systems. However, as explained, the file concept is reinterpreted in different ways in the WSN scope.

Settings	NOR (ex: ST M25P40, Intel PXA27x)	AT45DB	NAND (ex: Samsung K9K1G08R0B)
Erase	Slow (seconds)	Fast (ms)	Fast (ms)
Erase unit	Large (64KB-128KB)	Small (256B)	Medium (8KB-32KB)
Writes	Slow (100s kB/s)	Slow (60kB/s)	Fast (MBs/s)
Write unit	1 bit	256B	100's of bytes
Bit-errors	Low	Low	High (requires ECC, bad-block mapping)
Read	Bus limited[2]	Slow+Bus limited	Bus limited
Erase cycles	10^4 - 10^5	10^4	10^5 - 10^7
Intended use	Code storage	Data storage	Data storage
Energy/byte	1uJ	1uJ	.01uJ

Table 4.3: Flash memory chip settings

Operations associated with this kinds of devices are related to read and write pages of the flash memory, erase a page, or format the complete memory.

4.4.1.5. I/O serial bus

A sensor node (or gateway) could additionally offer serial communication capabilities to external devices. Examples of these sensor nodes are Telos family motes and ESB platforms, and mib520 gateways, all of them using a USB connection. It adds new features to sensor nodes, which at the same time must address the problem of transmitting and receiving data from the serial cable (both RS-232 as USB).

4.4.1.6. An example: MicaZ sensor node

The current subsection describes a MicaZ sensor node using the previous hardware specification. The goal is to validate if a specific instance could be represented. Consider the MicaZ sensor node composed of the MPR400 module (including microcontroller and radio) and the MTS300 sensor board. Figure 4.6 shows a component diagram in which the particular physical components integrated into the node, settings and functions are presented. In order to achieve component description of every type, a *Document Type Definition* (DTD)[3] must be elaborated.

[2]M25P40 reads are limited by the use of a 25M Hz SPI bus.

[3]It is important to point out that DTDs have been elaborated instead of XML Schemas for contents specification, due to practical reasons but nothing obligates to it. Furthermore, the process of transformation between DTD and XML Schema and vice versa is automatic using a proper conversion tool.

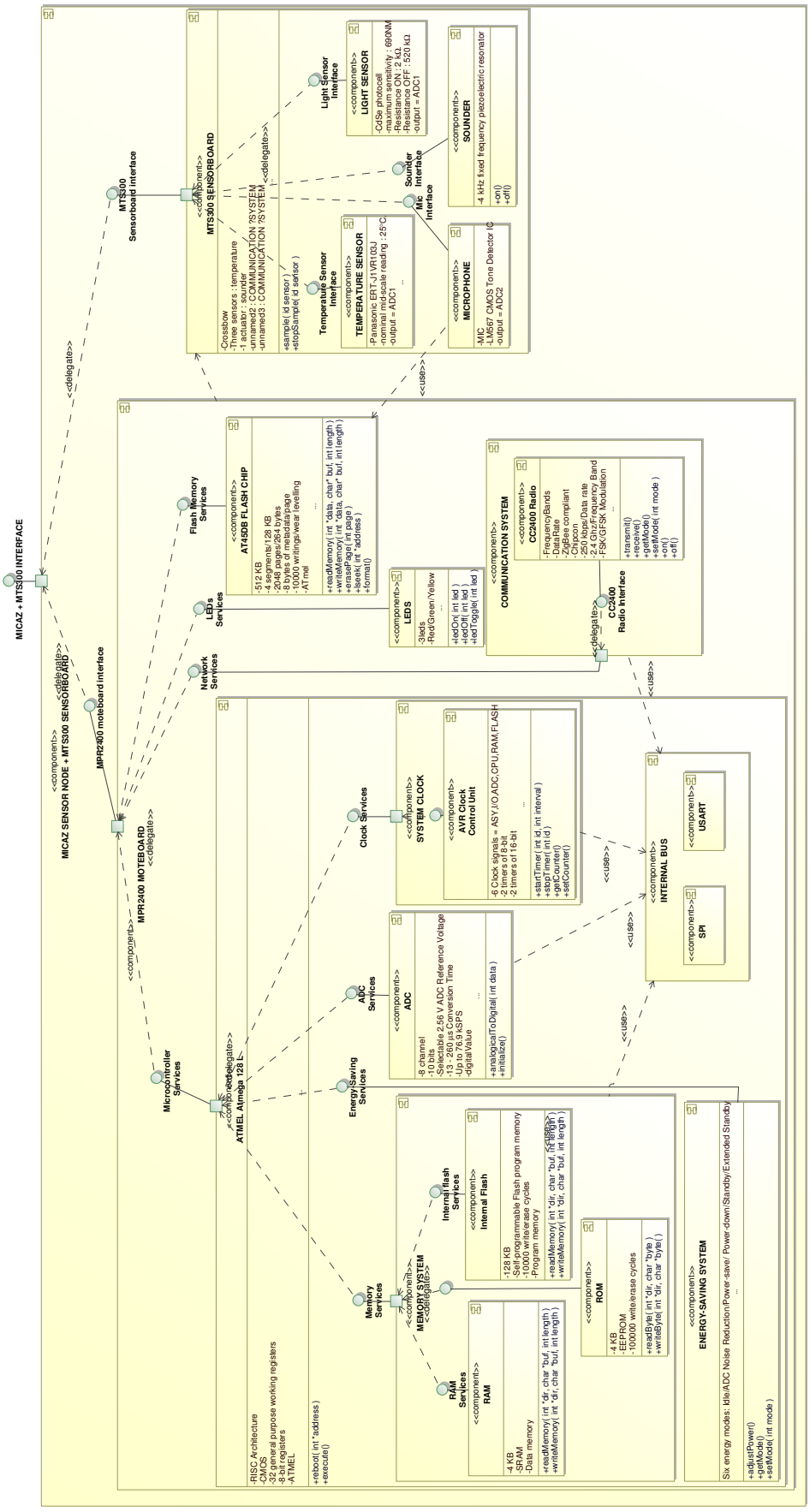

Figure 4.6: Implementation diagram of MicaZ sensor node with an MTS300 sensor board attached.

Note that the information contained in a DTD (or XML Schema) is that required by each component, and subsequently, it may be customized. Moreover, from this DTD its XML Schema can be generated automatically for thermistor sensors.

Consider the component *Temperature Sensor* in the figure. As mentioned, each type of physical device is characterized by an XML Schema or DTD, which collects information about the settings, interface and other details. A DTD for a thermistor sensor is presented in Listing A.1 from Appendix A. In spite of the fact that each kind of sensor (thermistor, light, microphone) or actuator (sounder) has different characteristics, it is possible to find similar features in the same kind of device. Listing 4.1 shows an XML Manifest in accordance with the previous DTD, describing the thermistor sensor integrated into the MicaZ platform.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SENSOR SYSTEM "Temperature_sensor.dtd">
<SENSOR>
  <Manifest Signature="ERT-J1VR103J-0000000004">
  <definition datasheet="www.panasonic.com/ERT-J1VR103J">
    <model>ERT-J1VR103J</model>
    <vendor>Panasonic</vendor>
    <type>Thermistor</type>
    <sensorboard>
      <board model="MTS300CA, MTS310CA"/>
    </sensorboard>
    <reading>
      <units name="degrees" out="digital"/>
      <scale size="25"/>
      <resistance ohms="" constant="" variable="Panasonic.txt"/>
      <value min="-40" max="70"/>
    </reading>
  </definition>
  <signal>
    <output name="ADC1"/>
    <location name="RT2"/>
    <power name="INT2"/>
  </signal>
  <assemblies>
    <file name="SensorTest" provider="Panasonic" fileAssembly="
       SENSOR/THERM/ERT-J1VR103J/Assemblies/test"/>
  </assemblies>
    <interface>
      <export>
        <operation>
          <action name="setOn" fileAction="SENSOR/THERM/ERT-
              J1VR103J/Operations/setOn.c" ReturnValue="error">
            <parameter index="0" type="signature"></action>
          <action name="setOff" fileAction="SENSOR/THERM/ERT-
              J1VR103J/Operations/setOff.c" ReturnValue="error">
            <parameter index="0" type="signature"></action>
          <action name="read" fileAction="SENSOR/THERM/ERT-
              J1VR103J/Operations/read.c" ReturnValue="error">
            <parameter index="0" type="signature"></action>
        </operation>
      </export>
    </interface>
```

```
    </Manifest>
</SENSOR>
```

Listing 4.1: Characterizing one thermistor sensor using an XML Manifest.

Consider also the component *MTS300 Sensorboard* in the figure. Listing A.2 in Appendix A shows the contents description of the MTS300CA Sensor Board [Tec] through its XML Manifest. Any model of sensor board (e.g. MDA100, MDA300) could be characterized also using the same XML Schema.

4.4.2. Operating System Layer instantiation

WSN operating systems are intended to efficiently manage the physical resources integrated into motes. OSes offer services to applications by mapping the interface provided by the hardware into higher-level services which are understandable by programmers. In this section these services are explored through three popular WSN operating systems: TinyOS 1.x, TinyOS 2.x and Contiki 2.2. The services are grouped into seven common categories, each one of them corresponding to one OS-specific functionality:

- *Local time services* orchestrates and synchronizes the operations implemented in the application. It virtualizes the clocks into the microcontroller in order to set different timers. In event-based operating systems and embedded devices, this functionality acquires more value because many operations are executed when the associated timer is fired.

- *Network services* establishes a network protocol stack for (wireless) communication capacity of sensor nodes. Different configurations should be taken into account (from ZigBee compliant to proprietary radios). It should include aspects related to the physical radio and the network protocols supported at the different levels.

- *Serial communication services* allow communicating the gateway device to a directly connected PC through a serial cable (e.g USB or RS-232). In this way, the gateway acts as bridge between the WSN and PC.

- *Flash memory services*. Typically, motes include an external flash EEPROM memory chip of small capacity for storing application data, such as aggregated values, sensor data or logs. Subsequently, this memory chip is susceptible to be used as a permanent storage device and therefore, it presents the relevant challenge of redefining sensor nodes-specific file systems.

- *Sensor services* mean the set of operations to manage the process of sampling the environment through sensors (readings), and additionally, producing certain outputs (e.g. sound) through actuators (writings) to the environment.

- *Debugging services*. LEDs devices provide a mechanism to debug applications in real scenarios because, correctly programmed, they can provide information about the state of a sensor node (for instance, red led on if failure condition).

- *Scheduling services* take decisions about the particular execution of the abstractions provided by the OS. Depending on the OS, these abstractions have different shapes and features: tasks (in TinyOS) or protothreads (in Contiki). Additionally, for a correct management, other services must be provided, such as blocking or synchronization primitives.

For each one of these functionality groups, the interface provided by each OS has been extensively studied, as shown in the next subsections.

4.4.2.1. TinyOS 1.x approach

This section studies the TinyOS 1.x (T1) interface offered to applications across the seven groups previously mentioned. Such interface will be termed $OI_{T1'}$, from now on. Note that not all existing interfaces have been collected. This means that a basic set of services to carry out a certain functionality has been considered. Assuming OI_{T1} as the complete interface from TinyOS 1.x, the previous one could be expressed in the followint way:

$$\begin{aligned} &OI_{T1'} \cap OI_{T1} \neq \emptyset \\ &\quad OI_{T1'} \subset OI_{T1} \end{aligned} \tag{4.19}$$

The next paragraphs will determine the $OI_{T1'}$ interface and its services. In Appendix B, the interfaces where such services are stated, are also identified, besides the components (both configuration and implementation) required by the application using these interfaces. Wirings to be included in the application are also presented. Finally, due to TinyOS forces to implement the events declared in the application interfaces, the list of events per interface is provided. It is important to point out that several alternative components implementing the interfaces could also be used, and subsequently different wirings would take place. The criteria to select components depends on different factors, such as the programmer experience or the abstraction level, preferably as high as possible.

- Time services: The microcontroller is able to individually manage a set of virtual timers (for example, in Mica family motes such a maximum number is fixed at 14). A timer needs to be initiated through the `start` command, which receives two arguments. The first one, `type`, is the mode in which the timer is started and it could take two values: `TIMER_REPEAT`, to get successive shots from a timer, and `TIMER_ONE_SHOT`, to get one single shot. The second one, `interval` establishes the interval for the timer expressed in binary milliseconds (1/1024 seconds). When the fixed interval expires, it is signaled to the application in the way of *event*. Table 4.4 shows the interface for timers in TinyOS 1.x.

Prototype	Interface	Description
command result_t start(char type, uint32_t interval)	Timer	Start a virtual timer
command result_t stop()	Timer	Preventing a started timer fired
async command tos_time_t get()	Time	Get the current time

Table 4.4: TinyOS 1.x interface for time services.

  In T1, adjusting the potency of the microcontroller is a platform specific operation. For ATMEL microprocessors, the interface *PowerManagement* is provided, but as a disadvantage, the computation of the new state must be done explicitly. MSP430 microprocessors are able to enter the lower power state with no explicit prompt. For peripherals devices, such as timers or the radio, the *StdControl* (or *SplitControl*) interface is typically used. The command `stop` causes that a subsystem to go into an inactive or low-power state. Table 4.5 collects some services to save energy.

- Network services provide the functionality for sending and receiving messages to and from the network via the radio device. The OS tries to abstract away every radio device, and

Prototype	Interface	Description
async command result_t enable()	PowerManagement	Enable a lower power mode
async command result_t disable()	PowerManagement	Disable the power mode
async command uint8_t adjustPower()	PowerManagement	Compute the low power state based on system configuration
command result_t init()	StdControl	Initialize component and subcomponents
command result_t start()	StdControl	Start component and subcomponents
command result_t stop()	StdControl	Stop component and subcomponents

Table 4.5: TinyOS 1.x interface for energy saving.

multiplex it through abstractions of communication denominated *Active Messages* (AM), as explained in Chapter 2. In this way, one application supports several virtual channels for independent communications both for sending and receiving. Every Active Message is labeled by an identifier which must be known by applications, in order to be explicitly specified in the corresponding wiring. Basically, network services provided by TinyOS 1.x are summarized in Table 4.6. Different implementations could use such an interface. In T1, the data packet is encapsulated into the `TOS_Msg` structure, which incorporates both the application data and the TinyOS header. It is important to point out that the action of receiving data from the network is implemented as an event, and therefore, it is never invoked by the application but by the operating system. If an application needs to process the received data, then it must provide the implementation of such event.

Prototype	Interface	Description
TOS_LOCAL_ADDRESS		The local address
TOS_BCAST_ADDRESS		The broadcast address
command result_t send(TOS_MsgPtr msg, uint16_t length)	Send	Send a message buffer
command void *getBuffer(TOS_MsgPtr msg, uint16_t *length)	Send	Get a pointer to a buffer
command result_t send(uint16_t address, uint8_t length, TOS_MsgPtr msg)	SendMsg	Send a message buffer

Table 4.6: TinyOS 1.x interface for network services.

- Serial communication is intended to communicate data from a sensor node or gateway to an external device, such as a PC or stargate, when both participants are directly connected using a serial bus (RS-232 or USB). It can be seen as a special case of network communication, where the target device is identified with the special address 0x7d. Therefore, network services can also be used for this feature, while the specific implementation is carried out by a different component (e.g. *UARTNoCRCPacket*, *UARTFramedPacket*), such as Table B.1 in Appendix B shows.

- Sensor services: TinyOS 1.x offers a single read interface for all sensors and a specific component for each one of them where the reading is implemented. Programmers use the interface *ADC* in order to sample a sensor, and the proper wiring to the implementation component must be explicitly done in the configuration component of application. When an analog sample has been converted to digital data, the system signals this fact through the `dataReady` event. The result of the reading is a 16-bit raw value, which must be transformed to engineering units using the proper formula. Samples must be taken in a sequential way, which means that a new sample cannot be initiated while there is a reading started. Table 4.7

shows the sensor interface.

Prototype	Interface	Description
async command result_t getData()	ADC	Initiates an ADC conversion
async command result_t getContinuousData()	ADC	Initiates a series of ADC conversions
command result_t init()	ADCControl	Initialize ADCControl structures
command result_t setSamplingRate(uint8_t rate)	ADCControl	Sets the sampling rate of the ADC
command result_t bindPort(uint8_t port uint8_t adcPort)	ADCControl	Remaps a port in the ADC portmap
command result_t enable()	ADCError	Enables error reporting
command result_t disable()	ADCError	Disables error reporting
command result_t muxSel(uint8_t sel)	Mic	Set the multiplexer's on (microphone)
command result_t gainAdjust(uint8_t val)	Mic	Set the amplification gain (microphone)
command uint8_t readToneDetector()	Mic	Return the binary tone detector's output

Table 4.7: TinyOS 1.x interface for sensor services.

- Debugging services: In TinyOS, when possible it is preferable to define a function per service and per argument instead a single function per service, whose behavior is discriminated by the value of a argument. In the words of Philip Levis: "*If a function has an argument which is one of a small number of constants, consider defining it as a few separate functions to prevent bugs*". In this way, LEDs devices present a specific primitive per led (red, green, yellow) and per service (on, off, toggle). Table 4.8 shows the interface for LEDs devices.

Prototype	Interface	Description
async command result_t redOn()	Leds	Turn the red LED on
async command result_t greenOn()	Leds	Turn the green LED on
async command result_t yellowOn()	Leds	Turn the yellow LED on
async command result_t redOff()	Leds	Turn the red LED off
async command result_t greenOff()	Leds	Turn the green LED off
async command result_t yellowOff()	Leds	Turn the yellow LED off
async command result_t redToggle()	Leds	Toggle the red LED
async command result_t greenToggle()	Leds	Toggle the green LED
async command result_t yellowToggle()	Leds	Toggle the yellow LED
async command result_t init()	Leds	Initialize the LEDs
async command uint8_t get()	Leds	Read the current LEDs information
aysnc command result_t set(uint8_t value)	Leds	Set a value into LEDs

Table 4.8: TinyOS 1.x interface for LEDs services.

- File systems. With the first version of TinyOS the Matchbox file system was distributed, as described in Chapter 2. Matchbox uses the flash memory chip for permanent storage of data. The interface offered is shown in Table 4.9. Matchbox supports only sequential reading and writing, and it allows opening several files simultaneously. However, the same file cannot be open for reading and writing. It also offers directory operations and other file operations such as renaming or deleting files.

- Scheduling services provide a basic management of concurrency at two levels: *tasks* and *events*, such as was explained in Chapter 2. Tasks in TinyOS 1.x are a form of *Deferred Procedure Call* (DPC) [CRH99], where a program defers a computation or operation until

Prototype	Interface	Description
command result_t open(const char *filename)	FileRead	Open a file for reading
command result_t close()	FileRead	Close file currently open for reading
command result_t read(void *buffer, filesize_t n)	FileRead	Read bytes sequentially
command result_t open(const char *filename)	FileWrite	Open a file for writing
command result_t close()	FileWrite	Close file currently open for writing
command result_t append(void *buffer, filesize_t n)	FileWrite	Write bytes sequentially to end
command result_t sync()	FileWrite	Ensure data appended is committed
command result_t start()	FileDir	List names of files found in the FS
command result_t readNext()	FileDir	Report next file name
command result_t rename(const char *oldName, const char *newName)	FileRename	Rename a file
command result_t delete(const char *filename)	FileDelete	Delete a file

Table 4.9: TinyOS 1.x interface for Matchbox file system.

a later time. Tasks encapsulate a small amount of processing, and are queued into a FIFO of 7 positions. nesC uses the next syntax to put and define tasks, as shown in Table 4.10.

Prototype	Description
result_t post *taskname()*	Put a task at the end of FIFO
task void *taskname(void)*	Prototype of a task

Table 4.10: TinyOS 1.x interface for scheduling services.

Let TinyOS 1.x (O_{T1}) be an OS expressed as the union of the complete set of components (C), wirings (W), and interfaces (OI_{T1}), thus:

$$\begin{aligned} O_{T1} &= \{C \cup W \cup OI_{T1}\} \\ OI_{T1} &= \{S \cup E\} \end{aligned} \tag{4.20}$$

and let (S) be the set of services (commands) offered and (E) the set of events included. OI_{T1} comprises the set of interface files including both services (S) and events (E). A TinyOS application is written using the nesC programming language (see Chapter 2). Subsequently, a TinyOS application requires a service (S) using the next notation:

$$\texttt{call } OI_{T1_i}\texttt{.S(Args)}$$

where OI_{T1_i} is the interface containing the service (S), which accepts the list of arguments (Args). The interface implementation is encapsulated into at least one implementation (or configuration) component (C). Additionally, the connection between the user and the provider of the interface OI_{T1_i} must be established. In terms of TinyOS, such a connection is denominated *wiring* (W). Both components and wirings must be included in the application configuration component, while used and provided interfaces are specified in the application implementation component. Following this interpretation, every service (S) collected in the Tables in Section B.3.1 (see Appendix B) is encapsulated in the interface OI_{T1_i}, which is implemented in the component (C) and also requires the wiring (W). Table B.1 in Appendix B shows the relation between a service, component(s),

wiring(s) and interface(s). Such relation could be expressed as:

$$\mathcal{F}(S) \rightarrow O_{T1},\ \forall S_i\ \exists\ \{c, w, OI_{T1_i}\} \subset O_{T1} \tag{4.21}$$

where $\mathcal{F}$ is the function to obtain the set of components, wirings and interfaces required for each service. For instance, consider the service `GreenOn` (S_i) belonging to $OI_{T1'}$. From Table B.1 in Appendix B, the following line is found:

Services	Interface	Component	Wiring
*	Leds	LedsC	X.Leds → LedsC

which means that any service exported by the interface *Leds* is implemented in component *LedsC*, and requires the wiring *X.Leds* → *LedsC*, where *(X)* corresponds to the application name. Note that only the component providing the service is shown. Additional components implied to complete the correct working (e.g. *StdControl*) are not specified.

When an application uses an interface, it is also forced to implement the complete set of events (E) exported by such interface. Table B.2 in Appendix B shows the list of events per interface. It is necessary to point out that the event declaration must include the name of the interface where it is located (prefixing the interface name and a dot to the service (S)). For example, taking as an example the `dataReady` event of the *ADC* interface, an application should rewrite its prototype in the following way:

```
async event result_t ADC.dataReady(uint16_t data)
```

On the other hand, applications provide both interfaces and their implementation, customizing in this way the default behavior given by system components. In particular, they must specify the implementation of every command included in the interface, and signal the occurrence of certain events to top-level components, using the following notation:

signal OI_{T1_i}`.E(Args)`

Consequently, it would be possible to express $OI_{T1'}$ as the union of the interfaces described in the current section, thus:

$$OI_{T1'} = OI_{T1'_{scheduling}} \cup OI_{T1'_{sensors\&actuators}} \cup OI_{T1'_{communication}} \cup OI_{T1'_{storing}} \cup OI_{T1'_{debugging}} \cup OI_{T1'_{clock\&energysaving}}$$

Figure 4.7 depicts an implementation diagram of TinyOS 1.x Operating system. It can be viewed as a specific instance from the general model shown in 4.4.

4.4.2.2. TinyOS 2.x approach

This section studies the interface offered by TinyOS 2.x (T2) to applications, which differs substantially from the previous version. Analogously to T1, this interface is named $OI_{T2'}$, from now onwards. Assuming OI_{T2} as the complete interface from TinyOS 2.x, it is possible to state:

$$\begin{aligned} &OI_{T2'} \cap OI_{T2} \neq \varnothing \\ &\quad OI_{T2'} \subset OI_{T2} \end{aligned} \tag{4.22}$$

Following, the $OI_{T2'}$ interface is extensively described.

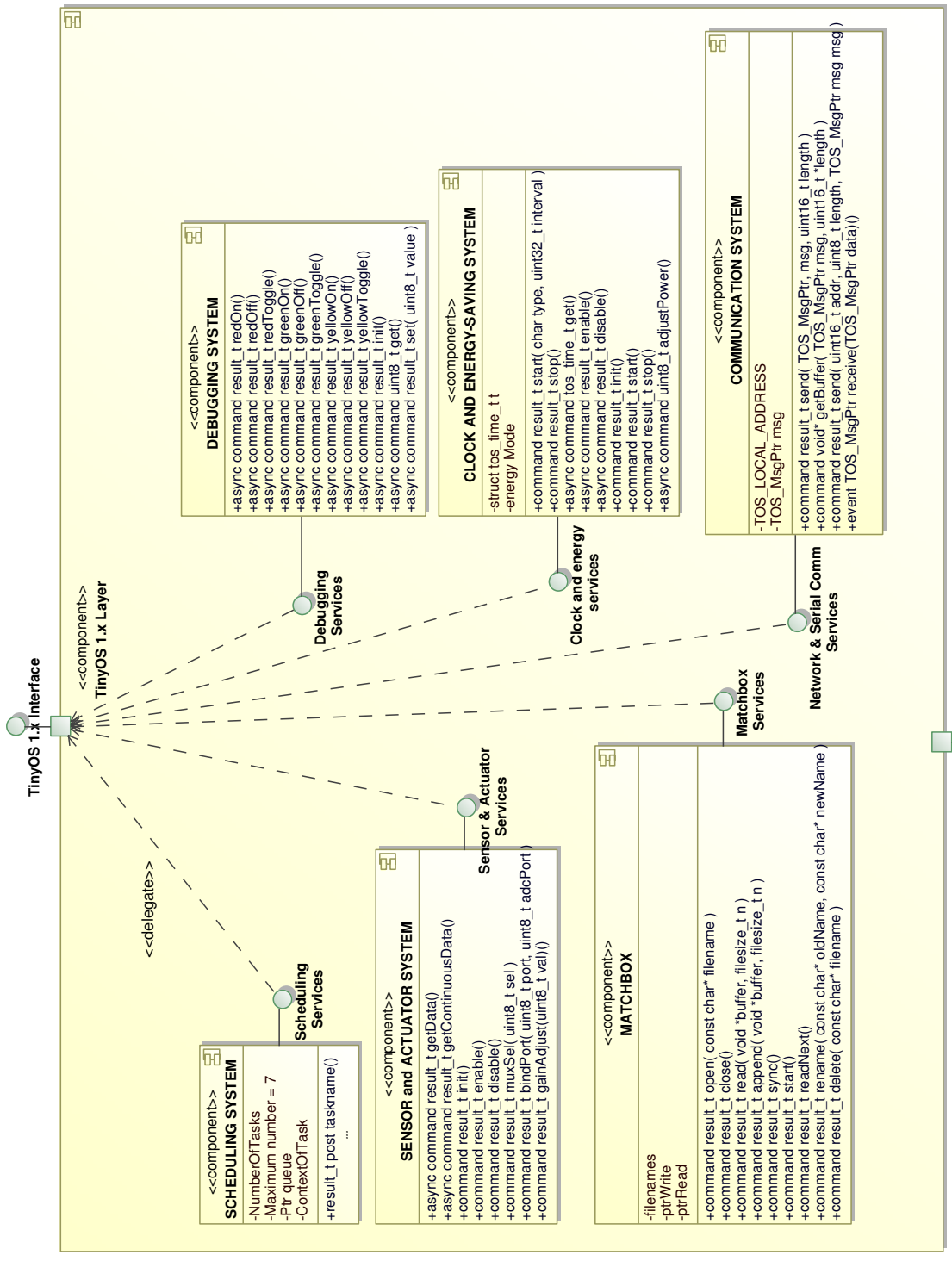

Figure 4.7: Implementation diagram of TinyOS 1.x.

- Time services: In the TinyOS 2.x approach the used component is determined for the timer precision. In this way, the configuration component *TimerMilliC* implies that the accuracy of the timer will be expressed in milliseconds. There are separated primitives for both modes of starting timers: single (`startOneShot`) or periodic (`startPeriodic`). As in TinyOS 1.x, when the timer expires, the corresponding event is signaled. Table 4.11 shows the services for timer management.

Prototype	Interface	Description
command void startOneShot(uint32_t dt)	Timer	Start a timer in dt time units
command void startPeriodic(uint32_t dt)	Timer	Periodically starts a timer every dt time units
command void stop()	Timer	Preventing a started timer fired
command uint32_t getNow()	Timer	Get the current time

Table 4.11: TinyOS 2.x interface for time services.

- For power saving management T2 offers more mechanisms than T1. For instance, *MCUSleep* takes the microcontroller to the *sleep* state, which will be woken up when an interruption arises. For devices power management, the interface *StdControl* is still maintained, in addition to *SplitControl* and *AsyncStdControl* interfaces. Table 4.12 depicts the interface for energy saving.

Prototype	Interface	Description
async command void sleep()	McuSleep	Put the microcontroller into a low power sleep state
async command void update()	McuPowerState	Compute the best low power state

Table 4.12: TinyOS 2.x interface for energy saving.

- Network services: Like TinyOS 1.x, TinyOS 2.x follows the communication paradigm based on *Active Messages*. However, the message structure has been modified (`message_t`), and the interfaces and components rewritten. The *AMPacket* and *Packet* interfaces are intended to access the fields of the packet, due to the fact that in the T2 header, the footer and metadata fields are opaque. Given that AM is a single-hop communication protocol, source and destination can be fixed through *AMPacket*. Other multi-hop routing protocols provide proprietary accessors. Table 4.13 shows the network services.
- Serial communication also uses the network interface previously described, while the implementation is encapsulated into the *SerialActiveMessageC* component, as shown in Table B.3 from Appendix B.
- Sensor services: In TinyOS 2.x the sensor and sensor boards devices (that is, the set of sensors attached to a sensor board) are represented through sensor drivers (HPL)[4]. The sensor devices are *generic components*[5] virtualizing the access to the sensor and connecting to the HIL. Sensor reading implies using the common interface *Read*. This interface takes as an argument the type of the data produced by the interface (typically an `uint16_t` value). Table 4.14 shows the interface for sensors.

[4] As explained in Chapter 2, hardware abstractions are classified into three groups (from bottom to top): *Hardware Interface Layer* (HIL), *Hardware Adaptation Layer* (HAL) and *Hardware Presentation Layer* (HPL).

[5] A generic component can be instantiated more than once.

Prototype	Interface	Description
command am_addr_t address()	AMPacket	Return the node's active message address
command am_addr_t destination(message_t *amsg)	AMPacket	Return the AM address of the destination
command am_addr_t source(message_t *amsg)	AMPacket	Return the AM address of the source
command void setDestination(message_t *amsg, am_addr_t addr)	AMPacket	Set the AM address of the destination field
command void setSource(message_t *amsg, am_addr_t addr)	AMPacket	Set the AM address of the source field
command bool isForMe(message_t *amsg)	AMPacket	Return if the specified message is destined to the mote
command am_id_t type(message_t *amsg)	AMPacket	Return the AM type of the AM packet
command void setType(message_t *amsg, am_id_t type)	AMPacket	Set the AM type field of the packet
command am_group_t group(message_t *amsg)	AMPacket	Return the AM group of the AM Packet
command void setGroup(message_t *amsg, am_group_t grp)	AMPacket	Set the AM type group of the packet
command am_group_t localGroup()	AMPacket	Return the current AM group of this interface
command error_t send(am_addr_t addr, message_t *msg, uint8_t len)	AMSend	Send a packet to the specified address
command error_t cancel()	AMSend	Cancel a requested transmission
command void clear(message_t *msg)	Packet	Clear out the packet
command uint8_t payloadLength(message_t *msg)	Packet	Return the length of the payload
command void setPayloadLength(message_t *msg, uint8_t len)	Packet	Set the length field of the packet
command uint8_t maxPayloadLength()	Packet	Return the maximum payload length
command void* getPayload(message_t *msg, uint8_t *len)	Packet	Return a pointer to payload region within a packet
command uint8_t payloadLength(message_t *msg)	Receive	Return the length of the payload
command void* getPayload(message_t *msg, uint8_t *len)	Receive	Return a pointer to payload region within a packet

Table 4.13: TinyOS 2.x interface for network services.

Prototype	Interface	Description
command error_t read()	Read<val_t>	Initiates a read of the value
command error_t read(uint32_t usPeriod)	ReadStream<val_t>	Starts to fill buffers by sampling with the specified period
command error_t postBuffer(val_t *buf,uint16_t count)	ReadStream<val_t>	Passes a buffer to the devices indicating its maximum length
async command error_t read()	ReadNow<val_t>	Initiates a read of the value

Table 4.14: TinyOS 2.x interface for sensor services.

- Debugging services: In the TinyOS 2.x approach, abstractions for LEDs devices are introduced as depicted in Table 4.15. The value of the LED 0,1 and 2 depends on the platform. In general, for each platform there is a mapping between the LED 0,1 and 2 into the red, green and yellow LED respectively. For example, the following sentences are located in the configuration component */opt/tinyos-2.x/tos/platforms/mica/PlatformLeds.nc*:

```
Led0 = IO.PortA2; //Pin A2 = Red LED
Led1 = IO.PortA1; //Pin A1 = Green LED
Led2 = IO.PortA0; //Pin A0 = Yellow LED
```

Prototype	Interface	Description
async command void led0On()	Leds	Turn on LED 0
async command void led1On()	Leds	Turn on LED 1
async command void led2On()	Leds	Turn on LED 2
async command void led0Off()	Leds	Turn off LED 0
async command void led1Off()	Leds	Turn on LED 1
async command void led2Off()	Leds	Turn on LED 2
async command void led0Toggle()	Leds	Toggle LED 0
async command void led1Toggle()	Leds	Toggle LED 1
async command void led2Toggle()	Leds	Toggle LED 2
async command uint8_t get()	Leds	Read the current LEDs information
async command void set(uint8_t)	Leds	Set a value into LEDs

Table 4.15: TinyOS 2.x interface for LEDs services.

- File systems: permanent storage on flash memory chip is reviewed in TinyOS 2.x. The concept of file in TinyOS 1.x is substituted by three high-level abstractions: *large objects* (e.g. programs received from network), *small objects* (e.g. configuration data) and *logs* (e.g. record-based data), which can be stored over partitions denominated *volumes*, whose size is configured at compilation time. Depending on the applications requirements, the programmer must select the more suitable abstraction. For each one of them, an interface is offered, as can be seen in Table 4.16.

- Scheduling and tasks management have been reviewed in the second version of TinyOS, such as it was described in Chapter 2. However, the interface for declaring and posting tasks is kept identical with respect to the first version.

Figure 4.8 shows the implementation diagram for TinyOS 2.x.

Programming rules previously stated for TinyOS 1.x are also valid for TinyOS 2.x. Analogously, the $\mathcal{F}$ function should be implemented to obtain the set of interfaces OI_{T2}, components (C) and wirings (W) associated with every service $S_i \in OI_{T2}$. In Appendix B, Table B.3 summarizes such relation. Events are deduced from Table B.4. Consequently, it would be possible to express $OI_{T2'}$ as the union of the interfaces described in the current section, thus:

$$OI_{T2'} = OI_{T2'_{scheduling}} \cup OI_{T2'_{sensors\&actuators}} \cup OI_{T2'_{communication}} \cup OI_{T2'_{storing}} \cup OI_{T2'_{debugging}} \cup OI_{T2'_{clock\&energy\ \ saving}}$$

Prototype	Interface	Description
command error_t erase()	BlockWrite	Erase the volume
command error_t write(storage_addr_t addr, void *buf, storage_len_t len)	BlockWrite	Write some bytes starting at a given offset
command error_t sync()	BlockWrite	Ensure all previous writes are present on a given volume
command error_t read(storage_addr_t addr, void *buf, storage_len_t len)	BlockRead	Read some bytes starting at a given offset
command error_t computeCrc(storage_addr_t addr, storage_len_t len, uint16_t crc)	BlockRead	Compute the CRC of some bytes starting at a given offset
command storage_len_t getSize()	BlockRead	Return bytes available for large object storage in volume
command error_t append(void *buf, storage_len_t len)	LogWrite	Append some bytes to the log
command error_t erase()	LogWrite	Erase the log
command error_t sync()	LogWrite	Guarantee that data written so far will not be lost to a crash or reboot
command storage_cookie_t currentOffset()	LogWrite	Return cookie representing current append position
command error_t read(void *buf, storage_len_t len)	LogRead	Read some bytes from the current read position
command storage_cookie_t currentOffset()	LogRead	Return cookie representing current read position
command error_t seek(storage_cookie_t offset)	LogRead	Set the read position to a value
command storage_len_t getSize()	LogRead	Return an approximation of the log capacity in bytes
command error_t mount()	Mount	Mount the volume
command error_t read(storage_addr_t addr, void* buf storage_len_t len)	ConfigStorage	Read some bytes starting at a given offset
command error_t write(storage_addr_t addr, void* buf storage_len_t len)	ConfigStorage	Write some bytes to a given offset.
command error_t commit()	ConfigStorage	Make the small object reflects all the writes since the last commit
command storage_len_t getSize()	ConfigStorage	Return the number of bytes that can be stored in the small object

Table 4.16: TinyOS 2.x interface for permanent storing.

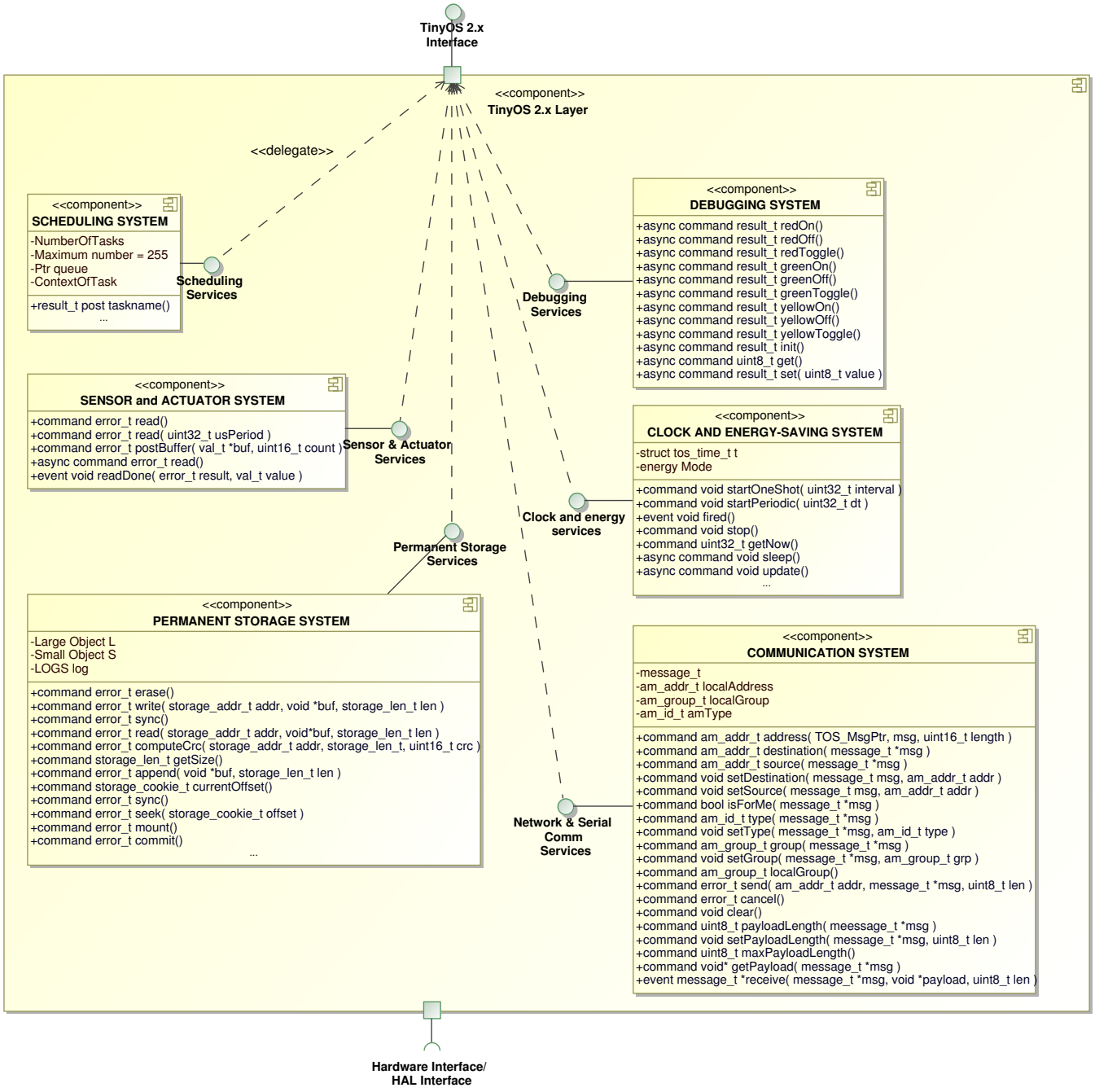

Figure 4.8: Implementation diagram of TinyOS 2.x.

4.4.2.3. The Contiki approach

In this subsection, the Contiki interface is classified following the same criterion as in the previous cases. Note that, in the same way as in TinyOS, a subset of the whole interface is shown, in order to limit the interface minimizing the functionality reduction. This interface is called

$OI_{Contiki'}$, and it can be expressed in the following way:

$$OI_{Contiki'} \cap OI_{Contiki} \neq \emptyset$$
$$OI_{Contiki'} \subset OI_{Contiki} \tag{4.23}$$

being $OI_{Contiki}$ the complete interface for Contiki OS. Due to the fact that Contiki is developed in C programming language, the applications use the conventional programming rules, except for the declaration and management of process, called *protothreads* (see Chapter 2). This means that, for instance, there is not a `main` function but an explicit declaration of the protothreads involved into the application.

- Time services: In Contiki, timers and event management are separated into two different libraries: Timer and Event timers. The first one is not able to signal applications that a timer has expired, but it must be manually checked. The event timer library carries out both tasks: managing timers and posting the events associated to applications. The structure supporting an event timer is denominated `etimer`. Applications must get a pointer to that structure in order to accomplish different tasks, such as starting and stopping a timer. Table 4.17 shows such interface.

Prototype	Description
void etimer_set(struct etimer *et, clock_time_t interval)	Set an event timer
void etimer_stop(struct etimer *et)	Stop a pending event timer
void etimer_restart(struct etimer *et)	Restart an event timer from the current time point
int etimer_expired(struct etimer *et)	Check if an event timer has expired
void clock_init(void)	Initialize the Clock libray
clock_time_t clock_time(void)	Get the current clock time

Table 4.17: Contiki interface for time services.

- Network services: As shown in Chapter 2, Contiki OS implements two communication stacks: *uIP* and *Rime*. The first one aims to provide Internet communication abilities to 8-bit microcontrollers, using a lightweight version of TCP/IP protocol suite. The second one is intended to offer a set of routing protocols implementing different features. For compatibility with TinyOS 1.x, which does not present up to date mechanisms for connecting individual motes directly to the global Internet and in spite of the fact that TinyOS 2.x incorporates a 6LowPAN implementation, the network services considered are focused on Rime stack (see Figure 2.11). Programmers must select the more suitable protocol from *Rime* stack depending on the application requirements. Table 4.21 depicts the basic primitives for only three *Rime* protocols: `abc`, `mesh` and `trickle`. Every routing protocol has a similar interface composed of three public functions: *open* and *close* a connection, and *send* a packet using that connection. Analogously, every routing protocol manages events (such as receptions or retransmissions) through *callbacks* or bottom-up signals produced among modules.

- Serial communication: Contiki offers a platform-dependent interface to directly connect from a sensor node to a PC. In this section, only ESB and Tmote SKY platforms have been

considered. The ESB platform holds a MSP430 microcontroller, which presents a RS-232 interface and whose implementation is located in the file *platform/cooja/dev/rs232.c*. On the contrary, the Tmote SKY board has an USB connector, whose implementation is provided in the file *cpu/msp430/dev/uart1.c*. Table 4.18 shows both interfaces.

Prototype	Description
void rs232_print(char *cptr)	Print a text string on RS232
void rs232_init (void)	Initialize the RS232 module
void rs232_set_input (int(*f)(unsigned char))	Set an input handler for incoming RS232 data
void uart1_init(unsigned long ubr)	Initalize the RS232 port.
void uart1_writeb(unsigned char c)	Put the outgoing byte on the transmission buffer
uint8_t uart1_active(void)	Return if R2-232 is transmitting or receiving

Table 4.18: Contiki interface for serial communication.

- Sensor services: Every sensor device has an specific implementation and is encapsulated into a header and implementation file, called *sensorname*-sensor.h and *sensorname*-sensor.c respectively, where *sensorname* is substituted by the sensor in particular: temperature, vib, pir, and so on (e.g. temperature-sensor.c, vib-sensor.c). Applications must include the header file corresponding to the sensor which it is going to use. The interface offered by the header files is common to the complete set of sensors. Readings from sensors return an `int` value. The sensor interface is shown in Table 4.19.

Prototype	Description
static void init(void)	Initialize the sensor
static int irq(void)	
static void activate(void)	Activate the sensor
static void deactivate(void)	Deactivate the sensor
static int active(void)	Check if the sensor is active
static unsigned int value(int type)	Get a sample from a sensor
static int configure(int type, void* c)	Configure the sensor

Table 4.19: Contiki interface for sensor services.

- Debugging services: In the Contiki approach, every LED is mapped to a constant value, as shown in the following lines:

```
#define    LEDS_GREEN     1
#define    LEDS_YELLOW    2
#define    LEDS_RED       4
```

  Contrary to TinyOS, Contiki uses these keywords to reference LEDs, and declares a common and shared interface to manage them. The interface for LED devices is presented in Table 4.20.

- File system: As described in Chapter 2, the Contiki file system is called Coffee. It uses a POSIX-like interface, as shown in Table 4.22. The first four primitives have been added to CFS (Contiki File System), the original version of the file system.

Prototype	Description
void leds_on(unsigned char leds)	Turn the leds LED on
void leds_off(unsigned char leds)	Turn the leds LED off
void leds_toggle(unsigned char leds)	Toggle LED leds
unsigned char leds_get(void)	Read the current LEDs information
void leds_arch_init(void)	Initialize LEDs
void leds_arch_set(unsigned char leds)	Set a value into LEDs

Table 4.20: Contiki interface for LEDs services.

- Scheduling services: As explained in Chapter 2, *protothreads* are the abstractions allowing the sequential flow of control for the Contiki processes, as opposed to the event-driven programming model.

  A Contiki process consists of a single protothread. Protothreads can be viewed as blocking events. The main difference between the two resides in the blocking capacity: protothreads can block (Contiki provides a wide set of synchronization primitives), while events cannot (in fact, TinyOS does not support any blocking abstraction). Protothreads and events are both stackless: all protothreads in a Contiki system use the same stack, while the same statement is valid for events in the TinyOS system. In relation to the protothreads scheduling, Contiki does not specify a policy to schedule protothreads, but such an approach is defined by the system itself: protothreads will run when the scheduled event sets it in the first place on a queue. Events can be asynchronous (causing the protothread to be scheduled some timer later) or synchronous (immediate scheduling). This scheduling differs from TinyOS, due to the fact that here the events are the functions with the highest priority and preempt both tasks as handler events, and in Contiki system they must execute to completion. Analogously, events in TinyOS reflect simple state transitions, while the processing is performed through tasks. Hardware interrupts both in the Contiki and TinyOS systems map events: timers, notification of data proceeding from a sensor, or a message received from the network. Contiki also provides a by default not-preloaded library in the kernel implementing preemptive multithreading. Scheduling services and declaration of protothreads can be viewed in Table 4.23. Note that they are focused on specifying only the services based on protothreads, for compatibility to TinyOS, which presents an execution model equivalent to protothreads.

Consequently, it would be possible to express $OI_{Contiki'}$ as the union of the interfaces described in the current section, thus:

$$OI_{Contiki'} = OI_{Contiki'_{scheduling}} \cup OI_{Contiki'_{sensors\&actuators}} \cup OI_{Contiki'_{communication}} \cup OI_{Contiki'_{storing}} \cup OI_{Contiki'_{debugging}} \cup OI_{Contiki'_{clock\&energy\ \ saving}}$$

Finally, Figure 4.9 shows the implementation diagram for Contiki operating system.

Prototype	Description
extern rimeaddr_t rimeaddr_node_addr	The Rime address of the node
void rimebuf_clear(void)	Clear and reset the Rime buffer
int rimebuf_copyfrom(const char* from, uint16_t len)	Copy from external data into Rime buffer
void abc_open(struct abc_conn *c, uint16_t channel, const struct abc_callbacks *u)	Set up an abc connection
int abc_send(struct abc_conn *c)	Send an anonymous best-effort broadcast packet
void abc_close(struct abc_conn *c)	Close an abc connection
void mesh_open(struct mesh_conn *c, uint16_t channel, const struct mesh_callbacks *u)	Set up a mesh connection
int mesh_send(struct mesh_conn *c, rimeaddr_t *dest)	Send a mesh packet
void mesh_close(struct mesh_conn *c)	Close a mesh connection
void trickle_open(struct trickle_conn *c, clock_time_t interval, uint16_t channel, const struct trickle_callbacks *u)	Set up a mesh connection
void trickle_send(struct trickle_conn *c)	Send a trickle packet
void trickle_close(struct trickle_conn *c)	Close a trickle connection

Table 4.21: Contiki interface for network services.

Prototype	Description
int cfs_coffee_reserve (const char *name, cfs_offset_t size)	Reserve space for a file
int cfs_coffee_configure_log (const char *file, unsigned log_size unsigned log_entry_size)	Configure the on-demand log file
int cfs_coffee_format (void)	Format the storage area assigned to Coffee
void * cfs_coffee_get_protected_mem (unsigned *size)	Point out a memory region that may not be altered checkpointing operations that use the file system
CCIF int cfs_open (const char *name, int flags)	Open a file
CCIF void cfs_close (int fd)	Close an open file
CCIF int cfs_read (int fd, void *buf, unsigned int len)	Read data from an open file
CCIF int cfs_write (int fd, const void *buf, unsigned int len)	Write data to an open file
CCIF cfs_offset_t cfs_seek (int fd, cfs_offset_t offset, int whence)	Seek to a specified position in an open file
CCIF int cfs_remove (const char *name)	Remove a file
CCIF int cfs_opendir (struct cfs_dir *dirp, const char *name)	Open a directory for reading directory entries
CCIF int cfs_readdir (struct cfs_dir *dirp, struct cfs_dirent *dirent)	Read a directory entry
CCIF void cfs_closedir (struct cfs_dir *dirp)	Close a directory opened with cfs_opendir()

Table 4.22: Contiki interface for Coffee file system.

Protothread declaration and synchronization	Description
#define PROCESS_BEGIN()	Define the beginning of a process
#define PROCESS_END()	Define the end of a process
#define PROCESS_WAIT_EVENT()	Wait for an event to be posted to the process
#define PROCESS_WAIT_EVENT_UNTIL(c)	Wait for an event to be posted to the process, with an extra condition
#define PROCESS_YIELD()	Yield the currently running process
#define PROCESS_YIELD_UNTIL(c)	Yield the currently running process until a condition occurs
#define PROCESS_WAIT_UNTIL(c)	Wait for a condition to occur
#define PROCESS_WAIT_WHILE(c)	Wait while the c condition occurs
#define PROCESS_EXIT()	Exit the currently running process
#define PROCESS_PT_SPAWN(pt, thread)	Spawn a protothread from the process
#define PROCESS_PAUSE()	Yield the process for a short while

Prototype	Description
CCIF struct process * process_current()	Return the current process
process_event_t process_alloc_event(void)	Allocate a global event number
void process_start(struct process *p, char *arg)	Start a process
void process_exit(struct process *p)	Cause a process to exit
int process_post(struct process *p, process_event_t ev, void *data)	Post an asynchronous event
void process_post_synch(struct process *p, process_event_t ev, void *data)	Post a synchronous event
#define PROCESS_CURRENT()	Get a pointer to the currently running process
#define PROCESS_CONTEXT_BEGIN(p)	Switch context to another process

Table 4.23: Contiki interface for scheduling services.

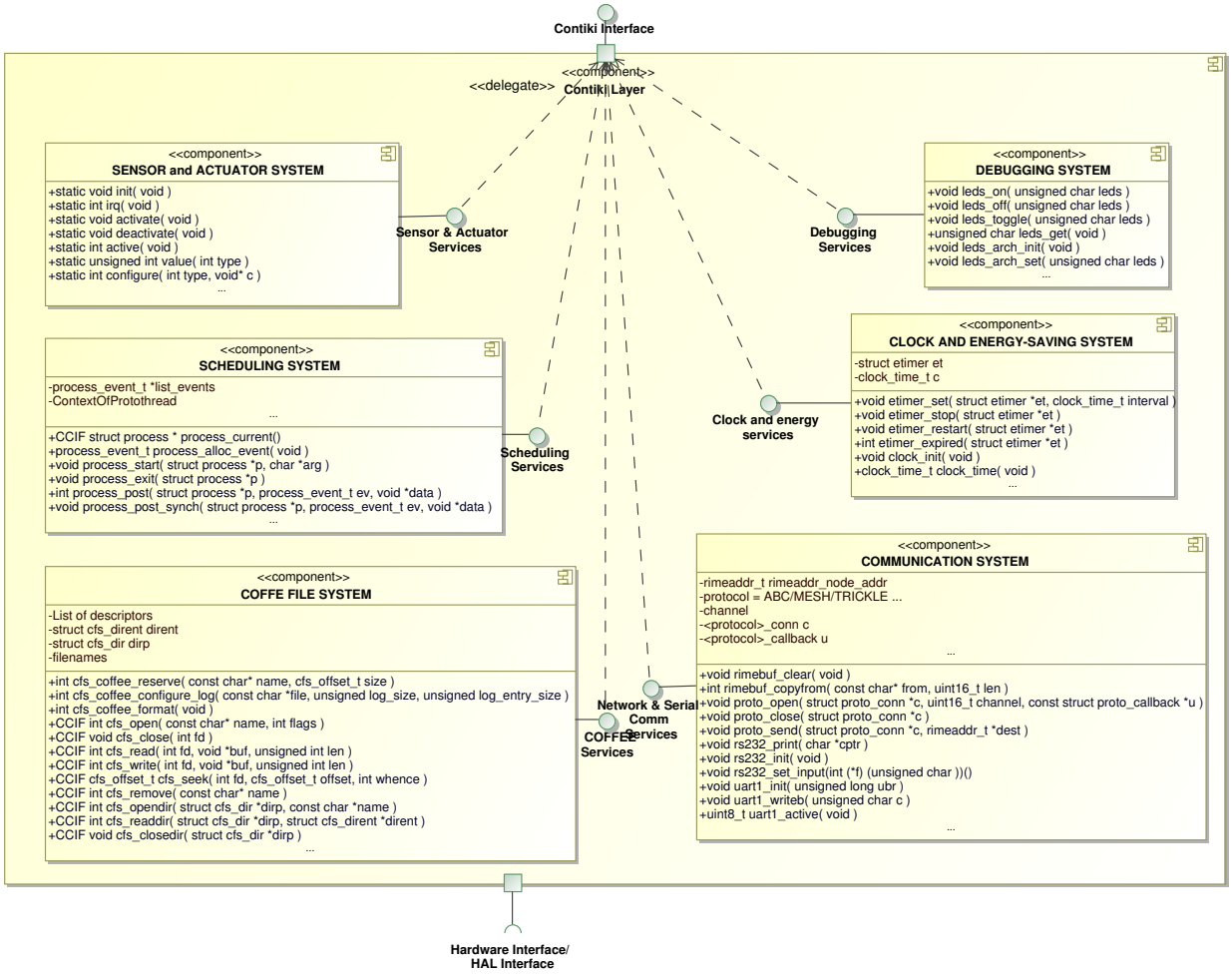

Figure 4.9: Implementation diagram of Contiki Operating System.

4.4.3. Operating System Abstraction Layer instantiation

This thesis proposes specifying and formalizing a flexible WSN architecture, in which developing WSN applications is simple. For it, a required step is to make the known components reusable, in order to incorporate new abstractions that deal with the problems of heterogeneity and complexity. As described, this abstraction has been denoted as *Operating System Abstraction Layer*, which is located on top of the OS. The goal is to facilitate the applications development, masking the underlying platform. For this purpose, an instance of it has been created: *Sensor-Node Open Services Abstraction Layer* (SN-OSAL). It follows the design specifications shown in Figure 4.5. OSAL establishes a DSL for homogenizing a set of WSN operating systems. Subsequently, the translation function denoted as $\searrow_T$, should be computed for them. Note that any instance of OSAL designed in accordance with such specification could be incorporated into the general design. Chapter 5 describes in depth the implementation details of SN-OSAL.

4.5. Extending the architecture

The last section of this chapter proposes studying the feasibility of extending the architecture proposed. This extension should be performed at the OSAL or OS level, including new functionalities or improving the existing ones. The reason for it could lie in several issues, for instance, an OS does not support a determined functionality (while the other does) or add distributed services (middleware). In particular, this section describes a contribution performed to improve the *Storage and File System* component in order to incorporate new services. Figure 4.10 shows the TinyOS 1.x design in which the Matchbox file system has been substituted by SENFIS file system. As shown, modifications only are necessary inside the implied component, in this case *Storage and File System*.

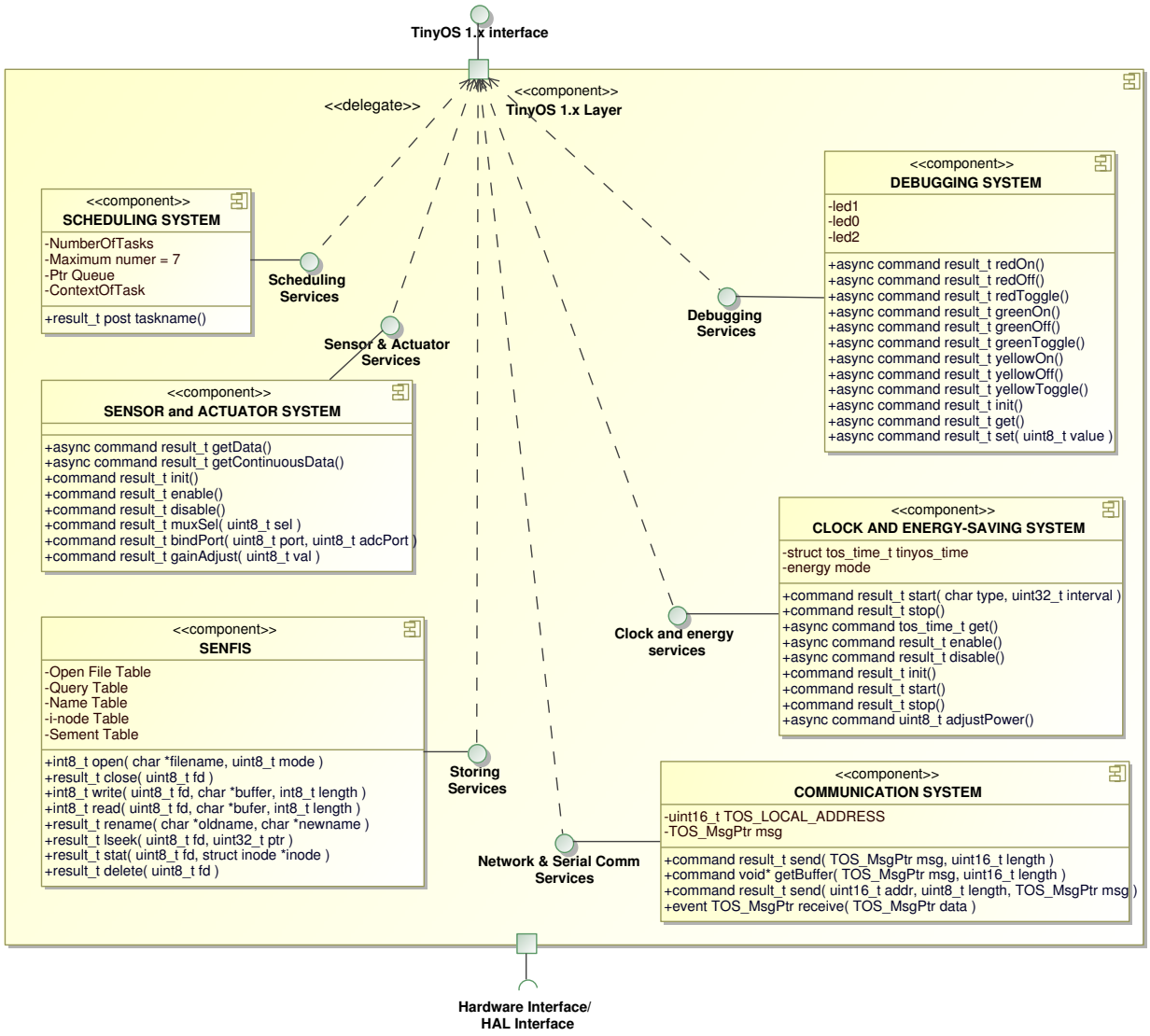

Figure 4.10: Implementation diagram of TinyOS 1.x including SENFIS.

SENsor node File System (SENFIS) is a novel file system for sensor nodes, which addresses

both scalability and reliability concerns. Design and implementation details of SENFIS are detailed. In particular, the SENFIS prototype uses the first version of TinyOS. In this way, SENFIS would substitute the original file system in T1, the Matchbox file system.

SENFIS can be mainly used in two broad scenarios. First, it can be transparently employed as a permanent storage for distributed TinyDB queries, in order to increase the reliability and scalability. Second, it can be directly used by a WSN application for permanent storage of data on the WSN nodes. The experimental section (see Chapter 8) shows that SENFIS implementation makes efficient use of resources in terms of energy consumption, memory footprint, flash wear leveling, while achieving execution times similarly with existing WSN file systems.

4.5.1. SENFIS Motivation

In [CK03] an overview of WSN applications is presented. The potential applications are classified into four categories: infrastructure security, environment and habitat monitoring, industrial sensing and traffic control. In the cases of the infrastructure security and industrial sensing, the direct approach consists of a sensor network connected by a fixed communication network with external (and unlimited) power sources. However, wireless networks could be used for providing more use flexibility and accesses. In these cases, external power sources could not be guaranteed. In the environment and habitat monitoring scenario, and partially, in the traffic control scenario, the sensor networks are configured using wireless *ad-hoc* networks with self-contained power sources. In these conditions, energy saving is critical for a large lifetime. Our approach targets this kind of configurations: the produced data is not necessary transferred out of the sensor node; instead, they can be temporary stored in the flash memory. Subsequently, the pieces of information associated with external queries or filtered by the mote internal application can be selected and sent through the wireless network to the central control unit.

In [GGP+03] the expected lifetime of a sensor network using centralized data collection is compared with the local file system approach for three equivalent scenarios. Results show a dramatic increase of the expected lifetime in about one order of magnitude with the use of a local file system.

4.5.2. Prerequisites: TinyDB

The WSN data selection process has been substantially facilitated by TinyDB [MFHH05], which is a distributed query processor running on the motes of a sensor network. TinyDB project focuses on acquisitional query processing techniques. Acquisitional query processing differs from other database query techniques for WSN in that it does not simply postulate the *a priori* existence of data, but it also focuses on location and cost of acquiring data. The acquisitional techniques have been shown to reduce the power consumption in several orders of magnitude and to increase the accuracy of query results.

A typical query of TinyDB is active in a mote for a specified time frame and is data intensive. Additionally, several queries may be pending in the same time. The results of a query may produce communication or be temporarily stored in the RAM memory. However, the motes of a sensor network have a limited memory. In some cases, due to hardware failures of environmental conditions, the radio network of a mote may become unavailable while the query is still active. Both in the case of large data sets production and of network failure it is important that enough memory space is available, more than the available RAM, and that the query data can be reliably stored on persistent storage.

In TinyDB the sampled values of the various sensor attributes (e.g. temperature, light) are stored in a table called `sensors`. The columns of the table represent the sensor attributes and the rows the instant of time when the measure was taken. Projections and transformations of the sensor table are stored in *materialization points*. A materialization point is a type of temporal table that can be used in subsequent select operations. Materialization points are declared by the users and correspond to files in our system.

TinyDB query syntax is similar to the `SQL SELECT-FROM-WHERE-GROUPBY` clause, supporting selection, join, projection and aggregation. In addition, TinyDB provides the `SAMPLE PERIOD` clause defining the overall time of the sampling called epoch and the period between consecutive samples. The materialization points are created by `CREATE STORAGE POINT` clause, associated with a `SELECT` clause, which selects data either from the sensor table or from a different materialization point.

A TinyDB query works in the following way: A user or an application propagates the query throughout the sensor network. The query reaches the motes, where it executes for the time specified in the `SAMPLE PERIOD` clause. The produced results stream to the root of the network, where they are gathered for processing.

4.5.3. System overview

SENFIS is a novel file system for motes, designed with the goal of increasing the scalability and reliability of TinyDB queries. SENFIS can be used in two scenarios, such as is shown in Figure 4.11. First, it can transparently be employed as a permanent storage for distributed TinyDB queries, in order to increase their reliability and scalability. Second, it can be as a stand-alone file system, for permanent storage of application data.

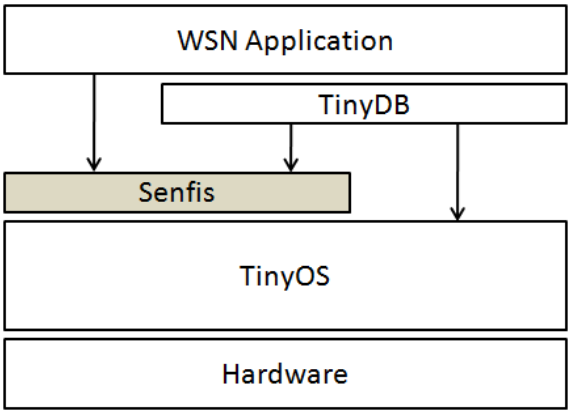

Figure 4.11: SENFIS architecture.

In the first case, SENFIS transparently stores query data and metadata in the case of RAM and network contention or if the network becomes temporary unavailable. TinyDB queries and materialization points are associated with files, which are persistently stored in the flash when the mote is put in the low-energy mode.

In the second case, SENFIS offers applications basic file and directory operations, which can be used for storing content in a persistent way.

SENFIS uses the flash for persistent storage and RAM as a volatile memory. The RAM

is used for run time file system structures. The flash is divided into segments, whose pages are accessed in a circular way, guaranteeing an optimal intra-segment wear leveling. The global wear leveling is a best-effort algorithm: a newly created file is always assigned the lowest used segment.

4.5.4. SENFIS design and implementation

This section presents the main data structures involved in the SENFIS implementation.

4.5.4.1. Flash data structures

The flash data structures of SENFIS are the *superblock*, the *segment table*, the *inode table* and the *file name table*. Figure 4.12 shows the organization of the metadata structures in SENFIS file system.

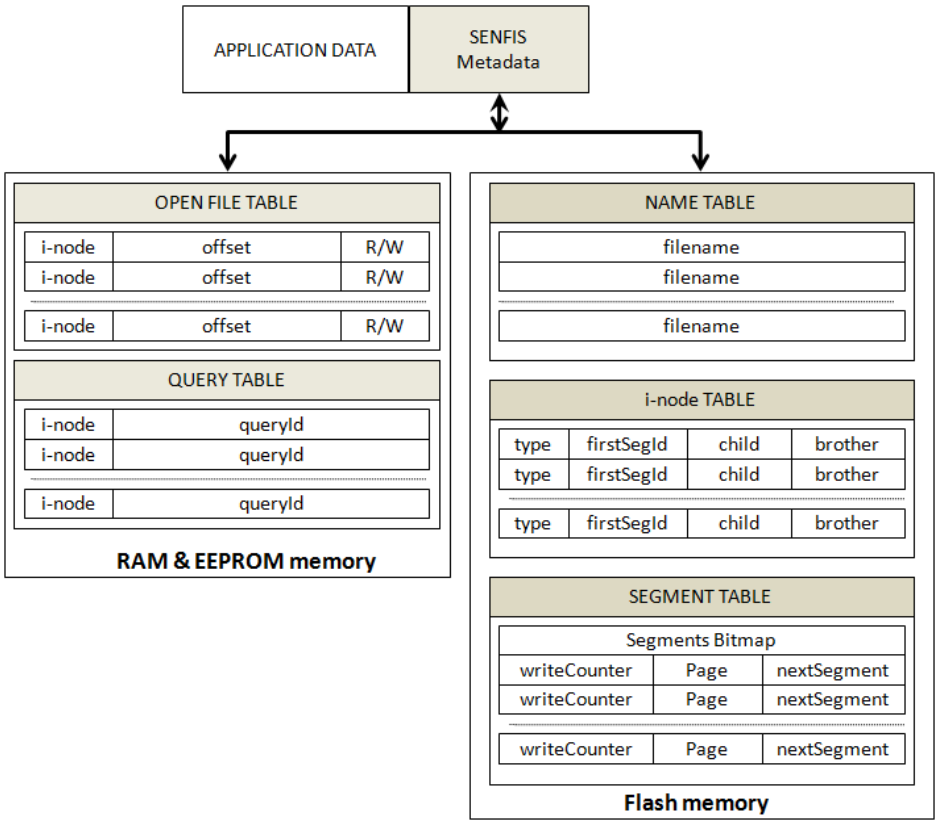

Figure 4.12: SENFIS organization of metadata structures.

The *superblock* is stored starting from the first byte of the first page of the partition. It contains metadata of the file system, bitmaps for inodes and segments and the necessary information for locating the other structures of SENFIS: the segment table, the inode table, and the file name table. A segment bitmap indicates which segments have been assigned to files and which segments are available. Similarly, the inode bitmap indicates, which inodes are assigned and which are available.

In SENFIS, the flash is organized into segments. For instance, for Mica2 the flash may consist

of 64 segments of 32 pages each. Each segment may be assigned to at most one file but a file can use an arbitrary number of segments.

A segment is always written sequentially in a circular way. The sequential writing corresponds to streams of data, the way queries and materialization points are generated. For implementing this behavior a pointer to the last written page is kept in the segment metadata structure which is stored in a *segment table*. The segment metadata contains the following fields:

```
struct segment{
   uint8_t writeCounter : 14;
   uint8_t firstPagePtr : 5;
   uint8_t nextSegmentID : 6;
}
```

The `writeCounter` indicates the number of times the pages of this segment have been written. The `firstPagePtr` is a circular pointer: when it reaches the end of the segment, the pointer is assigned again at the beginning of the segment and the write counter is incremented. In case a file to which the current segment was assigned is deleted, the current pointer value is kept for a future file assignment. In this way, a perfect wear leveling may be achieved inside each segment for the write streams such as the files generated by queries and materialization points. The `nextSegmentID` is used as an additional level of indirection for the files that require more than one segment.

In the case of Mica2 the total size of the segment metadata fits in a 256 bytes page: (14 bits + 5 bits + 6 bits) x 64 segments = 200 bytes.

The *i-node-table* is an array of the *i-node* structures. An inode structure stores the metadata of one file and contains the following fields:

```
struct inode{
   uint8_t type : 1;
   uint8_t firstSegmentId : 6;
   uint8_t brotherInode;
   uint8_t childInode;
}
```

The `type` indicates if the inode corresponds to a file or to a directory. The `firstSegmentId` identifies the first segment of the current file. The `brotherInode` and `childInode` are two inodes corresponding to the brother and child nodes in the name space tree.

The *file name table* is an array of file names of 8 bytes. This array is indexed by the inode number. The file names are separated by inode structures, in order to keep the inode structure compact and to concentrate the modifications of inode structures into a small number of pages.

4.5.4.2. RAM data structures

The RAM memory stores operational data structures used for SENFIS file access.

The *open file table* is an array of structures, containing information about the opened files. Each structure contains an inode identifier, the current file offset and a flag indicating if the file has been open for writing, reading or both.

The *query table* is an array of structures mapping TinyDB operations onto files. Each entry consists of a query identifier and an associated inode. An entry is created when a query reaches the mote and it is deleted when it finishes.

The RAM is used also for caching and updating flash structures (segment, inode and file name tables). The modifications of these structures are performed in RAM and the update process is performed periodically.

A sensor mote is typically active a few seconds each minute. In the remainder of the time, the mote is placed in a low-energy mode, in which the RAM content is lost. In SENFIS the RAM structures are backed up on EEPROM and are recovered on demand when the mote is activated.

4.5.4.3. SENFIS name space

SENFIS offers a hierarchical directory name space. The name space is embedded in the inode structure. The `brotherInode` points to a file or directory located in the same directory as the file represented by the inode structure. The `childInode` is used for non-empty directories: it points to an inode structure of a file or directory located in the directory of the current directory. Figure 4.13 shows an example of a SENFIS name tree implementation. The root inode contains three directories *a*, *b* and *c*. File *a* is associated with inode 0. The brother of *a* is *b* (inode 1) and its first child is *d* (inode 3).

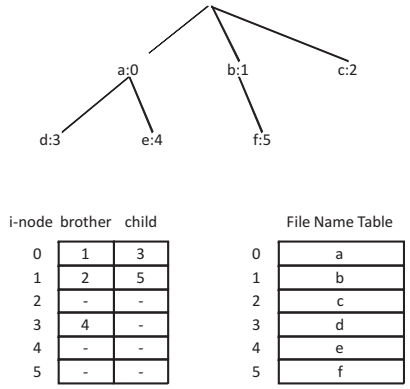

Figure 4.13: SENFIS name space example.

4.5.5. SENFIS operations

Table 4.24 shows the POSIX-like SENFIS API. This section describes only a subset of relevant SENFIS operations. Operations such as *stat*, *lseek* and *rename* have been implemented and are not discussed further here.

Note that these SENFIS primitives would constitute the interface of the *Storage and File System* component at the *Operating System Layer*, actually substituting the Matchbox component (see Figure 4.7).

Open. The open function uses the lookup operation for locating the file inode. The file path lookup is an internal operation employed by file open. It works in the following way: it traverses a height level of the tree looking for components among the right-brothers of a file/directory. If it does not find it the lookup fails. If a node is found, the lookup descends to the child and the search

Primitive Prototype	Description
int8_t open (char *filename, uint8_t mode)	Open a file
result_t close (uint8_t fd)	Close a file
int8_t write (uint8_t fd, char *buffer, int8_t length)	Append data to a file
int8_t read (uint8_t fd, char *buffer, int8_t length)	Read from a file
result_t rename(char *oldname, char *newname)	Rename a file
result_t lseek (uint8_t fd, uint32_t ptr)	Update the offset of a file
result_t stat(uint8_t fd, struct inode *inode)	Obtain metadata of a file
result_t delete (uint8_t fd)	Delete a file

Table 4.24: Basic high-level interface for SENFIS.

continues in the same way. The number of files in a mote is expectedly low, dozens of files and directories, therefore, the cost of this simple approach is low. If it is found, it retrieves the inode and reserves the first free available entry in the open file table of the corresponding partition and returns a reference to it (file descriptor). This file descriptor will be used for identifying the file in subsequent accesses. If the file does not exist, it is created. When a file is created, it adds a brother node to the right-most brother of a tree level, and therefore it implies the modification of an inode and the allocation of a free inode.

Close. The close function frees the entry in the open file table associated with the file.

Delete. File deletion updates the inode of the brother and father of the current directory and frees the current inode.

Write. The write operation appends data to the end of a file. The modification is done in a small buffer cache in RAM and it is committed to the flash either when a page is completely written or when the RAM is full. The first case tries to avoid a page being committed to flash several times for small writes.

Read. This operation reads the data from the flash to an application buffer. If the data is already in the small buffer cache, it is copied to the application buffer from there.

4.5.6. Using TinyDB and SENFIS

In the existing solutions the intermediary query results of TinyDB and the materialization points are not permanently stored on the flash memory. One of the usage scenarios of SENFIS is to transparently offer a permanent storage space for intermediary query results and materialization points.

This process works in the following way. When the RAM memory becomes full, or when the network transmission queue increases beyond a threshold, the query table is used and the query data is written to the corresponding SENFIS file. Later, either on demand for materialization points or upon availability of the radio network for queries, the query results are restored in the RAM, and the transmission is restarted by reinserting the query in the network transmission queue. The whole process is done transparently from the TinyDB users.

It can be pointed out that this approach contributes both to the scalability and reliability of TinyDB. As the flash memory is much larger than the RAM (for Mica2, 512 KB versus 4 KB), several queries can be executing in the same time. The reliability is increased in that in case of

outages of radio network, the query data is persistently saved, and, subsequently, restored upon recovery.

4.6. Chapter summary

In this chapter, the description of the sensor node-centric architecture has been presented. The architecture aims to mask the hardware heterogeneity to be able to write applications independently of the platform. This is carried out in a layered approach, which allows focusing on details of each architectural component. To achieve this goal, the deconstruction of the typical wireless sensor networks architecture, *Hardware* and *Operating System Layer*, has been treated as a mandatory step in its elaboration in order to be able to incorporate new abstractions and reuse the existing ones. Also, the overlying architecture has been designed in this step: the *Operating System Abstraction Layer*. The design is elaborated using implementation diagrams of UML 2.0. Following the design specifications, a mathematical description of the architecture is carried out. The formalization allows its representation with no ambiguity. In this way, the fundamentals of the architecture proposed are established.

Once this is performed, platform-independent models have been instantiated to be able to define particular instances of every one of the architectural layers previously identified. To describe physical devices integrated into sensor nodes, identifying its main features and operations, XML Manifests files are also used, according to the specification described through XML Schemas (or DTDs). It allows decoupling specification of components instantiation. Some examples have been presented in this chapter.

At the OS level, the interface has been extensively analyzed for three popular WSN operating systems: TinyOS 1.x, TinyOS 2.x and Contiki. Assuming these prerequisites, additional abstraction layers have been located on top of the traditional architecture: an *Operating System Abstraction Layer* and *Application Layer*, which constitute the main contribution of the thesis work and address the problem stated in Chapter 3. Finally, a preliminary description of *Sensor-Node Open Services Abstraction Layer* (SN-OSAL) is introduced as a specific instance of OSAL, the *Operating System Abstraction Layer*. The following chapters describe both layers in detail.

The work presented in this chapter has been previously published in [ECIG06, ECIC07, ECG+06].

Finally, the idea of extending the architecture at the OS level has been proposed, specifically, the file system SENFIS has been proposed. Preliminary work in file systems has been done in [ECIL08]. SENFIS has been previously published in the Journal of Supercomputing [EIM+09].

Chapter 5

Operating System Abstraction Layer (OSAL)

Wireless sensor networks have modified some traditional paradigms for software development and communications [EGHK99] [IGE00] [ASSC02], leading to certain interesting challenges for the research community [RKM02]. Due to the nature of sensor nodes and the special conditions of networks, WSN applications development is actually an *ad-hoc* process, which converts the applications composition into a complex task.

Currently, sensor node programming is a relevant challenge for the research community. Different paradigms have been described. At the network level there is an increasing trend to use middlewares to cover the gap between applications and networks [BC06] [CGG+05]. There are also approaches focused on the node level to hide the low-level details of the hardware platforms, such as WSN operating systems. However, existing solutions introduce coupling to the underlying operating system.

In the context of the proposed sensor node-centric architecture, this chapter aims to describe the proposed *Operating System Abstraction Layer*: *Sensor Node Open Services Abstraction Layer* (SN-OSAL). SN-OSAL is located on top of the most popular WSN operating systems (TinyOS 1.x and 2.x, and Contiki 2.2) in order to make applications programming independent from the underlying levels. OSAL masks complexity, and homogenizes access to lower levels, thus increasing the applications portability, which is the main goal of this thesis. To accomplish this task, SN-OSAL design is firstly specified as major components and interface. Secondly, it is demonstrated that SN-OSAL fulfills the specification proposed in the architecture formalized in Chapter 4. Implementation details are also widely described. As input SN-OSAL implementation should take generic applications written in accordance with the high-level DSL agreed upon and automatically generate the equivalent code for the selected target platform (operating system and hardware).

5.1. Sensor Node Open Services Abstraction Layer (SN-OSAL)

Assuming that systems have three abstraction layers (application, operating system and hardware), the proposal is based on the idea of introducing a lightweight abstraction layer between the application and the OS level. This abstraction layer is called *Sensor Node Open Services Abstraction Layer* (SN-OSAL). While other approaches model the hardware abstractions providing an API to operating systems, SN-OSAL is focused on homogenizing the OSes API to facilitate portable applications programming, and the writing of operating system-independent applications.

This means elevating the abstraction level of programmers with regard to other existing solutions. SN-OSAL design should clearly identify the functionality offered to the applications by OSes, the kind of functions that should remain hidden, and how this should be done.

5.1.1. SN-OSAL design

SN-OSAL is conceived as a services translator which makes the underlying architecture transparent to programmers. As explained, the interactions between these layers are performed through well-defined interfaces. SN-OSAL multiplexes the OS interfaces, encapsulating them into a single, homogeneous, generic interface called SN-OSAL API, which is exported to applications. Figure 5.1 presents a layered design of a sensor node-centric architecture in which SN-OSAL is located between applications and operating system. SN-OSAL is composed of two major components:

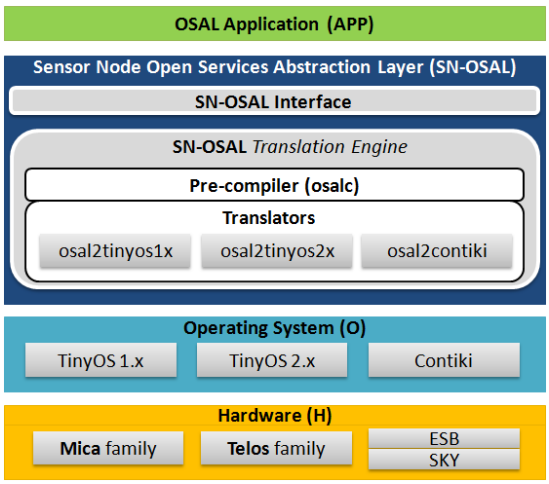

Figure 5.1: Sensor node-centric architecture using SN-OSAL.

- *SN-OSAL Interface* (SN-OSALI) provides a generic, portable, platform-independent and homogeneous POSIX-style programming interface to the upper layer. This interface consists of a set of high-level primitives to refer to basic services, reserved words, and a domain-specific language allowing the formal description of the applications with independence from the operating system. In particular, it should include abstractions for different operating system execution models, in order to hide the specific details to the applications.

- *SN-OSAL Translation Engine* acts as a kind of services demultiplexor and compiler, which receives a generic SN-OSAL application as input, and maps each of its services into operating system-specific requests, generating the operating system-specific application. In other words, this component is intended to perform the translation function previously described, denoted as $\diagdown_T : SN\text{-}OSALI \rightarrow OI_i$. As shown in Figure 5.1, it also consists of two sub-components: an applications pre-compiler called *osalc* and a set of translators.

The following sections deal with both major components in depth. Figure 5.2 depicts the SN-OSAL component following the design shown in Chapter 4.

In this way, an application developed on top of SN-OSAL can be translated into several equivalent platform-specific applications. WSN applications are written using the abstractions

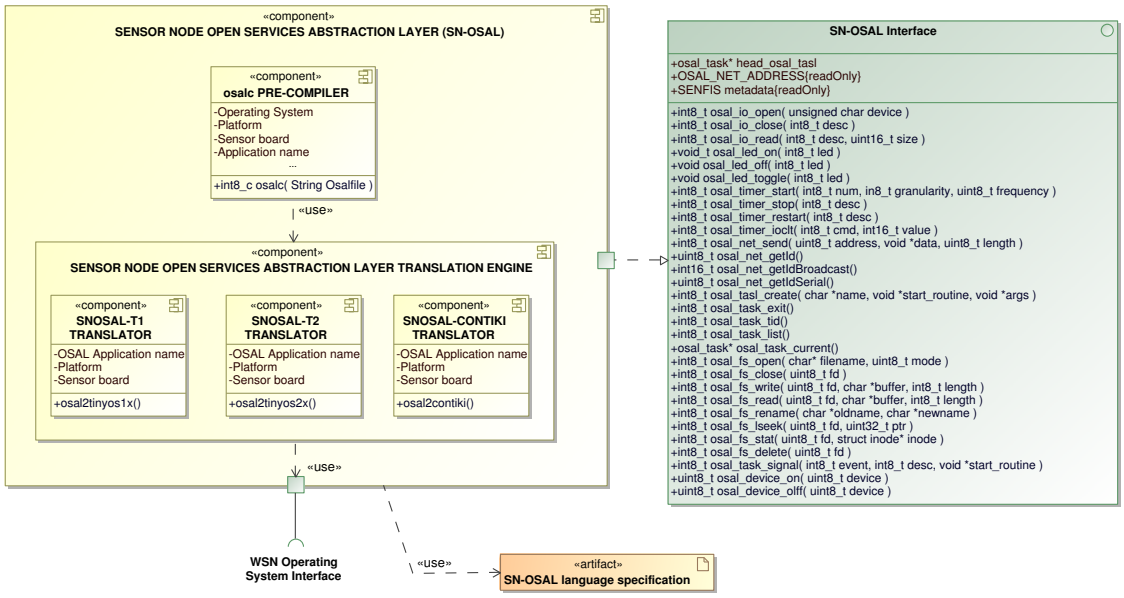

Figure 5.2: SN-OSAL description through an implementation diagram.

provided by SN-OSAL and, thus they do not directly access the native operating system interface. Subsequently, it is mandatory to formalize the writing rules (in addition to the interface) for the applications programming on top of SN-OSAL. The detailed proposal for applications building is dealt with in Chapter 6.

5.1.2. SN-OSAL formalization

Consider the sensor node-centric architecture formalization introduced in Chapter 4. In this subsection, this description is used to determine the feasibility of incorporating SN-OSAL as an instance of *OSAL* component.

Let SN-OSAL be an abstraction layer between applications and WSN operating systems, such that SN-OSAL ∈ OSAL, with OSAL being the set of all possible operating system abstraction layers defined on top of WSN operating systems (see 4.3). If SN-OSAL is an instance of *Operating System Abstraction Layer*, the subset of valid architectures using SN-OSAL $(A_{SN-OSAL})$ is a subset of **A** such that:

$$A_{SN-OSAL} = \{(a, SN\text{-}OSAL, b, c) \mid a \in APP \wedge b \in O \wedge c \in H, \\ (SN\text{-}OSAL, c) \in \mathcal{R}_T \circ \mathcal{R}_P\} \tag{5.1}$$

Portability achieved by SN-OSAL can be expressed as:

$$SN\text{-}OSAL \text{ supports } O_j \Longleftrightarrow SN\text{-}OSAL \, \mathcal{R}_T \, O_j \\ SN\text{-}OSAL \, \mathcal{R}_T \, O_j \Longleftrightarrow \exists \nearrow_T : SN\text{-}OSALI \rightarrow OI_j \tag{5.2}$$

To be able to extend SN-OSAL to the largest set of OSes, the cardinality of the next function must be maximized:

$$max(\mid (SN\text{-}OSAL, O_j) \in \mathcal{R}_T \mid) \text{ with } O_j \in O \tag{5.3}$$

Given that the SN-OSAL implementation currently supports two WSN operating systems, TinyOS (1.x and 2.x versions[1]) and Contiki, its interface ($SN\text{-}OSALI$) is exported to *Application Layer*, specifically:

$$OI_{T1} \cap OI_{T2} \cap OI_{Contiki} \subseteq SN\text{-}OSALI \subseteq OI_{T1} \cup OI_{T2} \cup OI_{Contiki} \tag{5.4}$$

In the current prototype, $SN\text{-}OSALI$ includes the set of common functions found in the OSes considered:

$$SN\text{-}OSALI = OI_{T1'} \cap OI_{T2'} \cap OI_{Contiki'} \tag{5.5}$$

where the interfaces $OI_{T1'}, OI_{T2'}, OI_{Contiki'}$ were stated in Chapter 4. Subsequently:

$$\begin{aligned} SN\text{-}OSALI &\searrow_T OI_{T1'} \\ SN\text{-}OSALI &\searrow_T OI_{T2'} \\ SN\text{-}OSALI &\searrow_T OI_{Contiki'} \end{aligned} \tag{5.6}$$

Consider $\mathcal{U}$ the universal set or maximum limit representing the complete interface ($\mathcal{U} = OI_{T1} \cup OI_{T2} \cup OI_{Contiki}$). Therefore, the set of primitives not included into SN-OSALI is expressed as $\mathcal{U} \setminus$ SN-OSALI:

$$\begin{aligned} &\mathcal{U} \setminus SN\text{-}OSALI = \\ &\{OI_{T1} \setminus SN\text{-}OSALI\} \cup \{OI_{T2} \setminus SN\text{-}OSALI\} \cup \{OI_{Contiki} \setminus SN\text{-}OSALI\} \end{aligned} \tag{5.7}$$

A design goal for SN-OSAL would be to minimize the set $\{\mathcal{U} \setminus$ SN-OSALI$\}$. As shown, currently SN-OSAL API covers the common functionality of all the operating systems studied. Therefore, SN-OSAL interface is minimal but, at the same time, a reasonable set of the supposed features has been collected.

Therefore, if O_j is a WSN operating system, to make applications transportable to O_j, a translation function is required ($\searrow_T$: SN-OSALI $\rightarrow OI_j$) (see Algorithm 4.1 in Chapter 4). That function is encouraged to match an SN-OSAL primitive with the IO_j-specific one. As mentioned, this connection is basically semantic, due to the fact that primitives in different abstraction levels represent the equivalent functionality. Note that this function must be established once for every OS supported by SN-OSAL.

Subsequently, it would be possible to determine the maximal and minimal functionality presented for OSAL as:

$$\begin{aligned} max(\searrow_T(SN\text{-}OSALI)) &= \bigcup \{OI_{T1}, OI_{T2}, OI_{Contiki}\}\ \{\text{T1, T2, Contiki}\}\ \in O \\ min(\searrow_T(SN\text{-}OSALI)) &= \bigcap \{OI_{T1}, OI_{T2}, OI_{Contiki}\}\ \{\text{T1, T2, Contiki}\}\ \in O \end{aligned} \tag{5.8}$$

Furthermore, let $APP_{SN\text{-}OSAL}$ be the set of potential applications written on top of SN-OSAL. Then:

$$\forall APP_{SN\text{-}OSAL_i} \in APP_{SN\text{-}OSAL}, APP_{SN\text{-}OSAL_i} = \bigcup x,\ \text{x}\ \in \{DSL_{SN\text{-}OSAL}\} \tag{5.9}$$

[1]The second version of TinyOS differs substantially from the first, particularly in application start up, task management and hardware access.

where $DSL_{SN\text{-}OSAL}$ is the DSL defined for programming SN-OSAL applications. Note that this DSL must include the SN-OSAL interface ($SN\text{-}OSALI$).

Finally, let $\mathcal{TP}$ be the function representing the transformation process which generates the equivalent OS-specific application from an SN-OSAL application, thus $\mathcal{TP}$: $APP_{SN\text{-}OSAL} \rightarrow APP_{OI_j}$. Formally, the process of generating specific applications is formulated as a composite function:

$$\mathcal{TP}(APP_{SN\text{-}OSAL}) = \begin{cases} \mathcal{K} \circ \mathcal{J} \circ \mathcal{H} \circ \mathcal{G}(APP_{SN\text{-}OSAL}) \text{ if } O_j = \{\text{T1, T2}\} \\ \mathcal{K} \circ \mathcal{J} \circ \mathcal{G}(APP_{SN\text{-}OSAL}) \text{ if } O_j = \{\text{Contiki}\} \end{cases} \tag{5.10}$$

where:

- $\mathcal{G}$ is the identity function represented as $\mathcal{G} : SN\text{-}OSALI \rightarrow SN\text{-}OSALI$. Its goal is to carry out the pre-compiling step of SN-OSAL applications.
- $\mathcal{H}$ is the function to obtain the set of OS-specific abstractions. Its goal is to obtain the components (C), wirings (W), and interfaces (OI_j) which must be included in the target TinyOS application. Note that it is TinyOS-specific. It uses the function $\mathcal{F}$ stated in 4.21.
- $\mathcal{J}$ is the function to obtain the set of event handlers to be included into the OS-specific application. Although in a different way, WSN OSes considered manage events in their applications.
- $\mathcal{K}$ is the function to perform the translation previously computed in $\diagdown_T$: $SN\text{-}OSALI \rightarrow OI_j$. Its goal is to translate the SN-OSAL code, and generate the equivalent OS-specific code.

The result of the $\mathcal{TP}$ function is the equivalent target application written using the specified OS. $\mathcal{TP}$ carries out the *transformation process* of WSN applications, according to the MDA standard. In following sections, these functions are described in detail.

5.2. SN-OSAL Interface (SN-OSALI)

The SN-OSAL subcomponent nearest to the application level is the *SN-OSAL Interface* ($SN\text{-}OSALI$). It means that *Application* and *Operating System Abstraction Layer* share the same programming abstractions, which allow operating system-independent applications to be written. As indicated, these abstractions basically conform a domain-specific language designed to express the functionality required by WSN applications (described in the next chapter).

The interface has been classified in primitives, constructs and keywords. Formally:

$$SN\text{-}OSALI = SN\text{-}OSALI_{Primitives} \cup SN\text{-}OSALI_{Constructs} \cup SN\text{-}OSALI_{Keywords} \tag{5.11}$$

where $SN\text{-}OSALI_{Primitives}$ is the set of primitives , $SN\text{-}OSALI_{Constructs}$ is the set of constructs, and $SN\text{-}OSALI_{Keywords}$ is the name space. Note that it must correspond with the domain-specific language used in the upper layer.

5.2.1. SN-OSALI primitives

SN-OSAL interface is composed of 30 POSIX-style primitives ($SN\text{-}OSALI_{Primitives}$), which can be used by the applications written over SN-OSAL. This set of primitives has been designed

with the aim to standardize the interface of the three OSes considered, presented in Chapter 4. These interfaces were denominated $OI_{T1'}$, $OI_{T2'}$ and $OI_{Contiki'}$, corresponding to T1, T2 and Contiki OSes respectively. Given their popularity, they are the most representative WSN operating systems. The interfaces considered for them manage all physical resources integrated into motes, and they also arise to be common among them. Therefore, the $SN\text{-}OSALI_{Primitives}$ set includes basic operations for writing applications. Note that the definition of this preliminary interface does not preclude of further extensions.

The interfaces are also classified according to the six functionalities considered: clock and energy saving, communication, debugging, sensors and actuators, scheduling and storing management. Note that the interface of SENFIS (see Chapter 4) has been encapsulated into calls for *Storage & File System* functionality. Table 5.1 lists SN-OSAL primitives. Appendix C shows the reference manual describing each *SN-OSALI* primitive.

5.2.2. SN-OSALI constructs

In addition to the set of primitives, it is necessary to define a set of *constructs*, which can be understood as *basic blocks of programming*. This set is named ($SN\text{-}OSALI_{Constructs}$). *Basic blocks of programming* means the programming language conventions that make writing syntactically correct applications possible.

The goal of these constructs is twofold: firstly, to formalize the syntax that SN-OSAL applications should exhibit and secondly, to encapsulate the high-level abstractions to map into the equivalent OS-specific *basic blocks of programming*. To reduce the learning curve, these constructs are also POSIX-based. The programming blocks and the corresponding constructs are defined below:

- *The main block* is represented by the `main` construct. This block represents the starting point of SN-OSAL programs. This function must be unique. Depending on the target OS, this initial point means generating different code templates.
- *Functions* are represented by the POSIX-style function prototype (analogous to C). Functions can be considered as high-level *tasks*, which are created using `osal_task_create` primitive, to encapsulate a certain functionality within an SN-OSAL program. Its goal is to favor the modularity of code.
- *Events* are represented in the same way as functions. The particularity is that they encapsulate the code for the event handlers written within programs. The primitive `osal_task_signal` must connect the signal with the function name to execute.

Through these constructs, the different programming blocks can be generated for the supported operating systems in an error-free way. Table 5.2 depicts the SN-OSAL constructs and their translations to Contiki and TinyOS.

5.2.3. SN-OSALI name space

Typically, the WSN applications development process includes tricky hardware details and device names. Every operating system defines its name space to refer to the physical devices from which they invoke certain services. As an example, consider the red led device. It is mapped to the logical name `LEDS_RED` in Contiki. In TinyOS, every operation over the red led is mapped to different services (`Leds.redOn, Leds.redOff, Leds.redToggle` in T1).

***Input and Output System* specific functions**
int8_t osal_io_open (unsigned char device);
int8_t osal_io_close (int8_t desc);
int8_t osal_io_read (int8_t desc, uint16_t size);
***Leds devices* specific functions**
void osal_led_on (uint8_t led);
void osal_led_off (uint8_t led);
void osal_led_toggle (int8_t led);
***Time & Energy saving System* specific functions**
int8_t osal_timer_start (int8_t num, int8_t granularity, uint8_t frequency);
int8_t osal_timer_stop (int8_t desc);
int8_t osal_timer_restart (int8_t desc);
int8_t osal_timer_ioctl(int8_t cmd, int16_t value);
uint8_t osal_device_on (uint8_t device);
uint8_t osal_device_off (uint8_t device);
***Network & Comm System* specific functions**
int8_t osal_net_send (uint8_t address, void *data, uint8_t length);
uint8_t osal_net_getId ();
int16_t osal_net_getIdBroadcast ();
uint8_t osal_net_getIdSerial ();
***Tasks and Scheduling System* specific functions**
int8_t osal_task_create(char *name, void * (*start_routine)(void *), void *args);
osal_task* osal_task_current();
int8_t osal_task_exit();
int8_t osal_task_tid();
int8_t osal_task_list();
uint8_t osal_task_signal(int event, int desc, void* (*eventhandler)(void*));
***Storage & File System* specific functions**
int8_t osal_fs_open (char *filename, uint8_t mode)
int8_t osal_fs_close (uint8_t fd)
int8_t osal_fs_write (uint8_t fd, char *buffer, int8_t length)
int8_t osal_fs_read (uint8_t fd, char *buffer, int8_t length)
int8_t osal_fs_rename(char *oldname, char *newname)
int8_t osal_fs_lseek (uint8_t fd, uint32_t ptr)
int8_t osal_fs_stat(uint8_t fd, struct inode *inode)
int8_t osal_fs_delete (uint8_t fd)

Table 5.1: *Sensor Node Open Services Abstraction Layer* primitives.

SN-OSAL interface specification also uses a set of logical names in order to access devices ($SN\text{-}OSALI_{Keywords}$). It is composed of reserved words, or keywords, to represent both physical components and arguments required by primitives.

OSAL construct	Contiki service	TinyOS 1.x service	TinyOS 2.x service

main	PROCESS(main, "main");
	AUTOSTART_PROCESS(&main);
	PROCESS_THREAD(main,ev,data) {
	... variables list ...
	PROCESS_EXITHANDLER(goto exit);
	PROCESS_BEGIN();
	//Implementation
		configuration *filename*{
		components Main, *filenameM*
		... components list ...
		// Wirings
		Main.StdControl->*filenameM*.StdControl;
		... wirings list ...
		}
		module *filenameM* {
		uses interface { ...}
		provides interface {StdControl,...}
		}
		implementation {
		command result_t StdControl.init()
		{ ...}
		command result_t StdControl.start()
		{ //Implementation
		}
		command result_t StdControl.stop()
		{ ...}
		}
			configuration *filenameAppC* {
			components MainC,*filenameP*
			... components list ...
			// Wirings
			filenameP->MainC.Boot;
			... wirings list ...
			}
			module *filenameP* {
			uses interface {Boot, ...};
			provides interface { ...}
			}
			implementation {
			event void Boot.booted() {
			//Implementation
			}
			}

function(y) && *osal_task_create(* *function,x,y)*	PROCESS_THREAD(function,ev,y) *... variables list ...* PROCESS_EXITHANDLER(goto exit); PROCESS_BEGIN();	task void *function()* *... variables list ...*	task void *function()* *... variables list ...*
} /*End of function*/	exit: PROCESS_END()		

function(args)&&
osal_task_signal(
eventname,y,z)
	PROCESS_WAIT_EVENT();
	if (ev == *eventname*){
	...event handler ...
	}
		event result_t *function(args){*
		...event handler ...
		}
			event result_t *function(args)*
			{ *...event handler ...*
			}

} /*End of event*/			

Table 5.2: SN-OSAL constructs and their translation to Contiki, TinyOS 1.x, and TinyOS 2.x operating systems respectively ($\nearrow_T : SN\text{-}OSALI \rightarrow OI_i$). From the SN-OSAL native code, the application template in every operating system is generated.

Devices must be named by applications using the reserved words for them, in order to easily and independently map into the specific device name. As in the case of operations, there is a connection between each OSAL keyword and the underlying operating system.

For instance, programmers should access the temperature sensor through the `TEMPERATURE` keyword, or to the red led through `LED_RED`. Analogously, additional settings have been defined, such as timer frequency (`ONE_SAMPLE, REPEAT_SAMPLE`), or its granularity (`SEC, MILLISEC`).

5.3. Translation function ($\rtimes_T$)

Once SN-OSAL and OSes interfaces have been identified, the semantical connection between functionalities in both levels must be established. This section presents this mapping. During SN-OSAL formalization, this function was referenced as $\rtimes_T$. The building of this function must be performed for every OS considered. Subsequently, the next three functions have to be computed for T1, T2, and Contiki respectively:

$$\begin{aligned} \rtimes_T &: SN\text{-}OSALI \rightarrow OI_{T1'} \\ \rtimes_T &: SN\text{-}OSALI \rightarrow OI_{T2'} \\ \rtimes_T &: SN\text{-}OSALI \rightarrow OI_{Contiki'} \end{aligned} \tag{5.12}$$

It means that a semantic association between SN-OSAL and these OSes should be found for every primitive in $SN\text{-}OSALI$, and, in this way generic high-level applications can be transparently translated into OS-specific ones.

In Appendix B, Tables B.5, B.6, and B.7 show the result of translating from SN-OSAL to T1, T2, and Contiki interfaces respectively. For every service in $SN\text{-}OSALI_{Primitives}$, the equivalent in every OS has been found. In the following sections, an example of translation is depicted for T1, T2 and Contiki OSes. During the translation process, which will be described later, these mappings are applied in order to generate OS-specific applications.

5.3.1. Mapping SN-OSAL to T1

Table B.5 in Appendix B shows the translation between the SN-OSAL and TinyOS 1.x interface. Thus, the function $\rtimes_T : SN\text{-}OSALI \rightarrow OI_{T1'}$ has been computed. Note that the TinyOS 1.x interface ($OI_{T1'}$) was determined in Chapter 4, in addition to the components, wirings, and interfaces, which are collected in Table B.1 in Appendix B. As an example, consider the following translations between SN-OSAL and T1:

OSAL primitive	T1 primitive	Platforms[2]
`osal_led_on(LED_RED)`	`call Leds.redOn()`	`All`
`osal_timer_start(*, SEC, ONE_SAMPLE)`	`call TimerX.start(TIMER_ONE_SHOT, \1 * 1000)`	`All`

[2]Due to the fact that not all primitives are portable to all platforms, the valid platform (sensor node or sensor board) for primitives is specified.

5.3.2. Mapping SN-OSAL to T2

Table B.6 in Appendix B shows the translation between the SN-OSAL and TinyOS 2.x interface. Thus, $\diagdown_T$: $SN\text{-}OSALI \rightarrow OI_{T2'}$ has been computed. In the same way as in the previous case, the TinyOS 2.x interface ($OI_{T2'}$) was established in Chapter 4, along with components, wirings, and interfaces, which are collected in Table B.3 in Appendix B. As an example, consider the following translations between SN-OSAL and T2:

OSAL primitive	T2 primitive	Platforms
`osal_led_on(LED_RED)`	`call Leds.led0On()`	`All`
`osal_timer_start(*, SEC, ONE_SAMPLE)`	`call TimerX.startOneShot( \1 * 1000)`	`All`

5.3.3. Mapping SN-OSAL to Contiki

Finally, Table B.7 in Appendix B shows the translation between the SN-OSAL and Contiki interface, that is, function $\diagdown_T$: $SN\text{-}OSALI \rightarrow OI_{Contiki'}$ is presented. Contiki interface ($OI_{Contiki'}$) was determined in Chapter 4. As an example, consider the following translations between SN-OSAL and Contiki operating system:

OSAL primitive	Contiki primitive	Platforms
`osal_led_on(LED_RED)`	`leds_on(LEDS_RED)`	`All`
`osal_timer_start(*, SEC, ONE_SAMPLE)`	`etimer_set(&tX, \1 * CLOCK_SECOND)`	`All`

5.4. SN-OSAL Translation Engine

Applications generation is accomplished by the *SN-OSAL Translation Engine*, which is the subcomponent of SN-OSAL intended to translate high-level applications written using the SN-OSAL interface into operating system-dependent applications. In the formalization described previously, the *SN-OSAL Translation Engine* performs the $\mathcal{TP}$ function. Conceptually, this process includes several steps, which can be summarized as follows:

1.- Pre-compile the SN-OSAL application in order to validate its correctness, in terms of accessing available devices and allowed functions in the target platform.

2.- Match and translate the SN-OSAL code into the OS-specific code. It may also be necessary to perform specific code adaptations, depending on the underlying OS.

3.- Automatically generate the platform-dependent compiling environment required.

4.- Compile and link to obtain the executable code, which can be downloaded into the target sensor node.

To accomplish these actions, the *SN-OSAL Translation Engine* is broken down into two different sub-components, as Figure 5.1 shows:

- A *pre-compiler* of SN-OSAL native code, called `osalc`. `osalc` verifies the applications source code in order to detect syntax or semantics errors. In the previous section, it is represented as function $\mathcal{G} : SN\text{-}OSALI \rightarrow SN\text{-}OSALI$.

- *Translators* intended to compile high-level applications (input) into OS-specific low-level code (output). Note that the translation function is inherent to the target OS. One translator must be created for each OS considered. They accomplish the $\mathcal{H}$, $\mathcal{J}$, and $\mathcal{K}$ functions described in 5.10.

5.4.1. The `osalc` pre-compiler ($\mathcal{G} : SN\text{-}OSALI \rightarrow SN\text{-}OSALI$)

The `osalc` pre-compiler performs the verification of the SN-OSAL native code, preventing badly-formed SN-OSAL applications. Such verification is both syntactic and semantic.

As input `osalc` takes a single properties file, which should specify the target platform where the application will be deployed. This file is called *Osalfile*, and it must include four lines specifying the following settings: sensor node platform (e.g. MicaZ, TelosB, SKY), sensor board (if required[3], e.g. micasb, mts300), WSN operating system for which the application will be generated (e.g. T1, T2 or Contiki), and, finally, the file name where the SN-OSAL application resides. Listing 5.1 presents an example *Osalfile*.

```
PLATFORM: mica
SENSORBOARD: basicsb
OPERATING_SYSTEM: tinyos1
OSAL_PROGRAM: /opt/tinyos1-x/apps/example.c
```

Listing 5.1: An example *Osalfile*.

Osalfile is passed as argument to the pre-compiler, which is invoked as follows:

```
./osalc <path_to_file>/Osalfile
```

`osalc` reads the *Osalfile* file to determine the platform and sensor board settings, and to locate the SN-OSAL application. Then, it sequentially parses every line of code of the SN-OSAL application in order to detect syntax errors. The source code of applications has to be written in accordance with the DSL format, which is described in the next chapter. Moreover, semantic control is accomplished: not allowed operations are also identified. Due to the knowledge of platform/sensor board settings, `osalc` can detect illegal hardware accesses. For instance, if the application tries to open an unavailable sensor in the selected platform, `osalc` would report the corresponding error.

After this verification process, a robust, error free, and syntactically correct SN-OSAL application is ensured, and `osalc` proceeds to invoke the corresponding translation script. On the contrary, if the application is not correct in terms of syntax or semantics, `osalc` aborts the process and provides an explanation error message to the user. In formal terms, function $\mathcal{G}$ has been defined as the identity function when pre-compiling an SN-OSAL application. Algorithm 5.1 summarizes the process associated with function $\mathcal{G} : SN\text{-}OSALI \rightarrow SN\text{-}OSALI$.

[3] In some platforms the capacity of processing and sensing is uncoupled.

Algorithm 5.1 `osalc`: Pre-compiling SN-OSAL applications

```
f(Osalfile) → platform, sensorboard, os, appname
open appname
error = 0
while (!error) and (!eof) do
   read line
   found = 0
   while (!found) and (∃ i ∈ SN-OSALI) do
      if i matches lines then
         if lines is valid (platform,sensorboard) then
            found = 1
      next i
   if (!found) then
      print error message
      error = 1
close appname
if os == tinyos1 then
   osal2tinyos1 (platform, sensorboard, appname)
else if os == tinyos2 then
   osal2tinyos2 (platform, sensorboard, appname)
else if os == contiki then
   osal2contiki (platform, sensorboard, appname)
else
   print error message
```

5.4.2. Translators ($\mathcal{K} \circ \mathcal{J} \circ \mathcal{H}$ ($APP_{SN\text{-}OSAL}$))

The *translators* are intended to carry out the translation and subsequent code generation from the verified SN-OSAL code obtained in the previous step. In formal terms, it implements the composite function integrated by:

$$\mathcal{K} \circ \mathcal{J} \circ \mathcal{H}(APP_{SN\text{-}OSAL}) \tag{5.13}$$

with its domain being the set of applications developed using SN-OSAL, and its codomain the set of equivalent operating system-specific applications (for T1, T2, or Contiki). For this purpose, one translator must be implemented for each OS. In this case, three scripts were done: `osal2tinyos1`, `osal2tinyos2`, and `osal2contiki` (Figure 5.1). Each script maintains a connection between SN-OSAL language statements (keywords, constructs and primitives) and the set of equivalent statements in the targeted operating system. This connection is provided by the translation function ($\nearrow_T$). In greater detail, the translation process includes:

1.- Obtaining the set of TinyOS-specific abstractions to be included in the equivalent application generated. In particular, it should compose the *configuration* component which includes required components, interfaces and wirings. This action has been denoted as $\mathcal{H}$ function.

2.- Generating the equivalent operating system-specific events from the high-level events handled in the SN-OSAL application. It means finding the event prototype corresponding to

TinyOS and Contiki, which both manage events in different way. This action has been denoted as $\mathcal{J}$ function.

3.- Each primitive and construct from the SN-OSAL interface must be translated into the set of operating system-specific equivalent services. It means generating the implementation files template, programming blocks, and the code to be properly inserted. Table 5.2 shows the translation of these constructs to each operating system. The correctness of these translations is discussed in Section 5.5. This action uses the mapping between SN-OSAL and the OSes ($\rightarrow_T$), and it has been denoted as $\mathcal{K}$ function.

The algorithm to obtain the equivalent application ($\mathcal{TP}(APP_{SN\text{-}OSAL}) = APP_{O_i}$) is summarized in Algorithm 5.2.

Algorithm 5.2 Translating SN-OSAL applications into OS-specific code

```
hw ← (Platform,Sensorboard)
os ← (Operating system)
if ((os == tinyos1) || (os == tinyos2)) then
   Create file ConfigurationFile_TinyOS
Create file ImplementationFile_os
for all i in APP_SN-OSAL do
   for all j in OI_os do
      if (∃ j such that →_T(i) = j) then
         print j ImplementationFile_os
         if ((os == tinyos1) || (os == tinyos2)) then
            F(j) = {C,W,OI_os}
            print {C,W} ConfigurationFile_TinyOS
            print {OI_os} ConfigurationFile_TinyOS
if !(error) then
   Create Makefile for {ConfigurationFile_TinyOS, ImplementationFile_OS }
   make
```

Perl scripting language was used to implement the translators. As described, translators carry out more work than a simple translation of primitives because they must generate functionally equivalent applications in spite of the different programming models. Thus, in order to completely convert the SN-OSAL program into the OS-specific code, other tasks must be accomplished, such as translating the constructs and events, which are expressed in a different way in each OS. Therefore, the transformation process is inherent to the OS. The next section describes the operations accomplished.

5.4.2.1. Obtaining TinyOS-specific abstractions ($\mathcal{H}$)

Specifically for TinyOS applications (both versions), the configuration component must also be generated. As described, the contents of this file include the whole set of components used by the application, and the interfaces and wirings among them. `osal2tinyos1` and `osal2tinyos2` translators manage the set of components, wirings, and interfaces which should be included for every SN-OSAL primitive. In this way, translators have to pre-process the SN-OSAL code to extract the configuration component.

This process is represented by the function $\mathcal{H}$ in previous sections. Let $\mathcal{H}$ be a function to obtain the set of components (C), wirings (W), and interfaces both used and provided (OI_i, *with* $i \in$

$T1, T2$) participating in the equivalent TinyOS application derived from the SN-OSAL application. Thus:

$$\mathcal{H} : APP_{SN\text{-}OSAL} \rightarrow \{C, W, OI_i, \varnothing\}$$
$$\mathcal{H}(APP_{SN\text{-}OSAL}) = \mathcal{F}(\succ_T(APP_{SN\text{-}OSAL})) \tag{5.14}$$

where $\mathcal{F}$ was defined previously (see 4.21) as the function to obtain the set of components, wirings and interfaces from TinyOS services. Tables B.5 and B.6 in Appendix B show this connection. Algorithm 5.3 presents the process $\mathcal{H}$: $APP_{SN\text{-}OSAL} \rightarrow \{C, W, OI_i, \varnothing\}$.

Algorithm 5.3 Obtaining the set of components, wirings, and interfaces

```
C, W, I={∅}
open APP_SN-OSAL
while (!eof) do
   read line
   for all (i ∈ SN-OSALI) do
      if i matches line then
         ≻_T(i) = j, where j in { OI_T1' ∪ OI_T2' }
         F(j)= {c, w, OI_os}
         C = C ∪ c
         W = W ∪ w
         I = I ∪ OI_os
close SN-OSAL
```

Consider the `osal_led_on(LED_RED)` service. To obtain the set of components, wirings, and interfaces involved, first it is necessary to find the associated TinyOS service, in this case `call Leds.redOn()`. Once the service is matched, it is necessary to obtain the set of components, wirings, and interfaces through function $\mathcal{F}$, as explained in Chapter 4.

5.4.2.2. Generating low-level events ($\mathcal{J}$)

As mentioned, SN-OSAL events must be translated to low-level events in the format of the underlying operating system. Events are managed in a different way in the WSN operating systems considered because their execution models follow opposite models, and, subsequently, the programming models use distinct mechanisms to express them. For instance, event-based model of TinyOS, system interfaces declare the events prototype that applications are forced to handle when this interface is used. Therefore, an SN-OSAL event can be mapped to a unique TinyOS event.

In the case of Contiki, the implementation done distinguishes between network and other events. In the first case, a similar mechanism to TinyOS is used, as network events are expressed using the event-based OSes traditional format. In the second case, there are no event handlers, but synchronization primitives are used to allow block or unblock in function of program conditions. This section describes how low-level events are inferred.

TinyOS The set of events handled in a TinyOS application can be analogously deduced from the SN-OSAL code. To declare one event handler, the `osal_task_signal` primitive must be invoked in order to specify an association between the event generated and the function to be executed. Its prototype is the following:

```
osal_task_signal (int event, int desc, osal_sighandler_t handler)
```

where `event` is the event name to manage, `desc` is the device on which this event will be signaled, and `handler` is a structure which includes the function name to be executed when the event is signaled (*event handler*). For the first argument, SN-OSAL keywords were defined for each event. Tables B.2 and B.4 in Appendix B show the relation between events and the interface files where defined, for T1 and T2 respectively. Consider the following prototype of function:

```
osal_task_signal (OSAL_IO_READDONE , TEMP, readDone)
```

where `OSAL_IO_READDONE` represents the event signaled when a sensor reading has been completed and data is available, `TEMP` identifies the device that waits for the event, and `readDone` is the event handler. To generate the equivalent event in TinyOS, an SN-OSAL service involving that device must have been invoked (in the example, a temperature sensor reading). This operation represents the operation over the device which relates the event to be managed. Note that the device might be both a variable and a keyword. For instance, for the `OSAL_IO_READDONE` SN-OSAL event, the events `dataReady` and `readDone` are deduced through the *ADC* TinyOS 1.x and *Read* TinyOS 2.x interfaces. The second argument identifies the interface name (or alias) required to complete the equivalent prototype. For instance, `TEMP` device represents the interface `ADC as Temp` and `Read as Temp` in accordance with Table B.1 and B.3 in Appendix B, for T1 and T2 respectively. Subsequently, the original prototype is rewritten for T1 and T2 as:

```
async event result_t Temp.dataReady(uint16_t data)
event void Temp.readDone(error_t result, val_t value)
```

Additionally, consider the function $\mathcal{J}$ intended to obtain the set of events associated with a specific interface. Thus, $\mathcal{J} : APP_{SN-OSAL} \rightarrow \{E, \varnothing\}$, where E is the set of events to be implemented within the application. This connection can be deduced from Table B.2 and B.4 in Appendix B. Algorithm 5.4 shows the process for $\mathcal{J}$: $APP_{SN\text{-}OSAL} \rightarrow$ {E, $\varnothing$}.

Algorithm 5.4 Generation of event handlers (TinyOS)

E={$\varnothing$}
OI_{os}=$\mathcal{H}(APP_{SN\text{-}OSAL})$
open $APP_{SN\text{-}OSAL}$
while (!eof) **do**
  read *line*
  for all ($s \in OI_i$) **do**
    if (s matches *line*) **then**
      if ($\exists$ osal_task_signal for s) **then**
        $EVENT \leftarrow$ First argument of *osal_task_signal*
        $DEVICE \leftarrow$ Second argument of *osal_task_signal*
        $HANDLER \leftarrow$ Third argument of *osal_task_signal*
        e = Compose prototype($EVENT$,$DEVICE$)
        $E = E \cup e$
      else
        print *error message*
close $APP_{SN\text{-}OSAL}$

Contiki Event translations from SN-OSAL applications to the Contiki OS are mainly related to the translation from the event-based to threads-based paradigm. It means that a blocking syntax is required and an iterative mechanism to drive the application must be provided.

For the first point, when a split-phase operation is invoked, an event handler will be immediately encoded. This event handler is composed of a blocking primitive and the code of the event handler, which will be executed when the condition for blocking disappears. The blocking syntax is obtained through active wait implemented by different Contiki macros, such as `PROCESS_WAIT_EVENT`. When an interruption occurs, the program continues on the next line which must evaluate whether the event it is waiting for has been signaled. If a handler was written for it in the SN-OSAL application (through `osal_task_signal`), then it is immediately executed to completion. When it finishes, then protothread is forced to resume its execution. Note that during the wait period, the scheduler can assign CPU time to any protothread according to its policy. The following lines show the template for Contiki events generation:

```
PROCESS_WAIT_EVENT();     /* Wait for whatever event */
if (ev == eventname)      /* Is it the expected event ? */
{
--EVENT_HANDLER--         /* Yes, so go with the handler */
}
```

Listing 5.2: Event handler generation in Contiki.

For the second point, applications driving, every protothread is maintained in an infinite loop to ensure that the application runs forever. Within this loop, the operations corresponding to every protothread should be located, while events management is performed as previously described. As mentioned, the context switch among protothreads is performed by the scheduler according to its policy.

Special attention is needed by network events management (e.g. receptions, acknowledgments, retransmissions). As explained, *Rime* (see Chapter 4) manages these events through *callbacks*, which can be viewed as network event handlers. These handlers are implemented only if the application requires that some processing be done (analogously to T1 and T2). *Callbacks* have a syntax similar to:

```
const static struct Y_callbacs X_callbacks = { X_recv };
static struct Y_conn X_conn;

static void X_recv(struct Y_conn *c) {
osal_comm_message osal_pkt;
memcpy (&osal_pkt, rimebuf_dataptr(), rimebuf_datalen());
--EVENT_HANDLER--
}
```

Listing 5.3: Callback generation in Contiki (network event handlers).

In Listing 5.3 X must be replaced by the name of the application, and Y must be replaced for the *Rime* protocol name used (due to the generality, any *Rime* protocol could be used). As shown, a network connection must be open, which by convention is denominated `X_conn`. The prototype of the reception event is `X_recv`, and the set of callbacks is denominated `X_callbacks`. Analogously to the reception, other network events can be described.

Subsequently, the $\mathcal{J}$ function to obtain events from the SN-OSAL application, has been rewritten for Contiki OS. Algorithm 5.5 shows how Contiki events are extracted from SN-OSAL applications.

Algorithm 5.5 Generation of event handlers (Contiki)

```
open APP_SN-OSAL
while (!eof) do
  read line
  for all (s ∈ SN-OSALI) do
    if s matches line then
      if (∃ osal_task_signal for s) then
        EVENT ← First argument of line
        DEVICE ← Second argument of line
        HANDLER ← Third argument of line
        if (EVENT is a network event) then
          Generate callback(APP_SN-OSAL)
        else
          Generate event handler
        Copy event handler code to –EVENT_HANDLER–
close APP_SN-OSAL
```

5.4.2.3. Translating primitives ($\mathcal{K}$)

The $\mathcal{K}$ function carries out the translation between SN-OSAL and the equivalent OS-specific primitives. To achieve this, it needs the previously computed $\rangle_T$ function (see Tables B.5, B.6, and B.7), which semantically relates them.

This process uses regular expressions and patterns. As a general rule, each SN-OSAL primitive is considered to be a pattern. Patterns can be literals, reserved words or special symbols such as "*". During the translation, the goal is to treat matching sentences in SN-OSAL programs with these patterns (the first column of each Table mentioned). If it matches, then the sentence of the SN-OSAL application will automatically be substituted by the service in the second column of the table, which corresponds to the semantically equivalent service in the selected OS.

Consider the table of mappings from SN-OSAL to T1 from Section 5.3.1. The simplest case appears in the first line: if a statement in an SN-OSAL program matches `osal_led_on(LED_RED)`, it will simply be substituted by `call Leds.redOn()` in the target code.

However, in many cases the searching pattern is unknown, and it is also necessary to recover this information from the arguments within SN-OSAL programs for later usage in the target application. In this case, the symbol "*" is used. As an example, consider the second line from the same previous table. In this case three patterns were used: "*", which represents the interval, and expresses whatever character is matched; `SEC`, which indicates timer granularity, and `ONE_SAMPLE` indicating timer frequency. Note that whatever character (or characters) written in the first argument would match pattern "*". These values are also remembered when they are referenced through the notation "\x" (see the second column), where "x" means the number of order of the pattern within the expression.

Variables management Code adaptations must be done in order to translate from SN-OSAL to OS-specific applications, for example related to variables management. Although SN-OSAL

allows descriptors to be defined, to identify resources (e.g. timers, files, sensors), TinyOS accesses different devices through customized interfaces, which must be explicitly included to invoke operations on them. These interfaces are usually identified through a distinguishable name, typically through an *alias*. In this way, if an application needs three timers, it must include three instances of the interface (e.g. Timer) with different aliases (`Timer0, Timer1, Timer2`), as follows:

```
uses interface Timer as Timer0
uses interface Timer as Timer1
uses interface Timer as Timer2
```

In this way, this function infers a correlation between the variable name assigned to one device in SN-OSAL, and the interface required to reference it in TinyOS, through an order number calculated from 0 to `n-1` (with `n` being the number of devices of one type), which is concatenated to the interface name (e.g. `Timer0`). For example, if three timers are declared in an SN-OSAL program, each one should be properly substituted into the specific code using the three previously mentioned aliases.

Unlike TinyOS, Contiki uses a syntax that is more similar to POSIX and defines variables to represent devices in the same way as SN-OSAL does: timers (from Clock), sensors, even files. Subsequently, it is possible to find a immediate correlation between device description in OSAL and Contiki. Given such correlation, the translation of SN-OSAL to Contiki variables is trivial.

5.5. SN-OSAL programming model discussion

The SN-OSAL programming model goals are to hide the underlying operating system specific code and to provide homogeneous way of developing WSN applications. Consider the different programming models shown by TinyOS and Contiki. TinyOS uses the nesC programming language, which defines a few reserved words to express things that we are not able to do in C (such as events or wirings). Contiki uses C macros to identify protothreads and their corresponding execution code.

A basic idea presented in this chapter is the proposal of a set of SN-OSAL constructs masking the programming paradigm and execution model of every OS. As shown, at a pre-compilation phase, these constructs are parsed by each script, and are substituted for the platform-specific code. These abstractions consist of three constructs (see Table 5.2) allowing the `main` function, execution entities, and event handlers to be represented.

The `main` construct identifies the main block of the application, in other words, the `main` function of a C program. For generating the equivalent TinyOS application, the source SN-OSAL application must be previously parsed in order to deduce the components, interfaces and *wirings* required for the configuration file[4]. Once this has been done, the implementation file is created. It must include the code described within the `main` block, which indicates the starting point of the application. For example, for TinyOS 1.x the starting point of the application is the command `start` of the interface *StdControl*, which is invoked by the operating system at start-up time. For TinyOS 2.x, the equivalent starting point is the event `Booted` of the interface *Boot*, which is invoked in same way by the operating system. Analogously, in the Contiki model programming, the programmer implements a set of functions (protothreads) specifying their initial and

[4]In TinyOS, the application is normally split into two files: a configuration file and an implementation file. The first collects the components used by the application, and the way they are related (wirings). The implementation file encapsulates its functionality.

final point through the PROCESS_THREAD and PROCESS_END macros. The execution sequence of these protothreads is given to the operating system through the AUTOSTART_PROCESSES macro. Consequently, the main block will always be the first protothread specified in that list. However, it is important to point out that there is no a specific scheduling and, execution is only depending on blocking conditions.

Execution entities can be started through the osal_task_create primitive. In order to carry out some control of the execution entities created (for instance, list the tasks or get its identifier), a new library has been implemented that is called *OS_SchedServices*, which cannot be mapped to any other. As mentioned, osal_task_create schedules a new SN-OSAL task, which means that an execution entity in the underlying operating system is created, and it can be either a protothread in Contiki or a task in TinyOS. The SN-OSAL task prototype is declared to be a typical C function, which identifies the starting point and where its execution code must be located. This task only starts to run when the underlying operating system schedules it: in TinyOS, when there are no pending events to execute (just like a task); in Contiki, when the current running protothread gets blocked in a condition and yields the microcontroller (just like a protothread). The end of this function must be translated in Contiki, because the protothreads are explicitly finished through the PROCESS_END macro.

Special attention is needed for the events management. TinyOS and Contiki manage events in very different ways. TinyOS explicitly defines the event handlers associated with each hardware interruption. Operations are generally split in-phase, which means that applications require services and as a response, when data is available, the hardware produces an interruption. Next, the operating system performs an up-call to the event handler, stopping any task in execution. When the event handler has finished, the task is resumed. In Contiki, only network events are managed through event handlers via *callbacks*. Blocking is explicitly done through different macros such as PROCESS_WAIT_EVENT or PROCESS_YIELD_UNTIL, allowing the protothreads to be blocked or letting other protothreads execute. When the hardware interruption occurs, the blocked process is unblocked if the condition is evaluated as false, and the handler associated with the event is executed (if defined). To provide an event-based programming model to applications, two actions are required: declaring the association between signal and handler through the osal_task_signal primitive; and encoding this event handler. This event handler prototype is univocally matched to the corresponding event in TinyOS. Note that different event names exist depending on the TinyOS version. In Contiki, an event manager is created, as shown in Table 5.2. This event manager blocks the application through the PROCESS_WAIT_EVENT macro until one event occurs. When this happens, the ev variable declared within the protothread arguments list identifies the event received and data identifies the associated result. The code within the handler is the code located within the SN-OSAL event described, and it is parsed as explained previously.

In conclusion, SN-OSAL respects the scheduling policies carried out by the underlying OS. The generated applications by SN-OSAL will be functionally identical, and the code is adapted to the specific OS. In fact, the SN-OSAL goal is not to produce equivalent applications in terms of execution, but in terms of syntax and semantics.

5.6. Chapter summary

In this chapter SN-OSAL has been proposed as a specific *Operating System Abstraction Layer* intended to be part of the proposed sensor node-centric architecture, and respecting the design presented in previous chapters. The chapter has dealt with SN-OSAL design and imple-

mentation in depth.

The high-level design of SN-OSAL has been presented, identifying its major components. Feasibility of integrating SN-OSAL into the proposed architecture has been probed in formal terms. Given the formalization provided in Chapter 4, SN-OSAL has been introduced as an element belonging to *Operating System Abstraction Layer*, fulfilling the specifications. Therefore, it guarantees that SN-OSAL can be integrated into a traditional WSN architecture.

The interface for SN-OSAL applications has been described as the intersection between the interfaces of the OSes considered, plus a set of constructs which can be translated to the basic blocks for applications building in every OS, as well as a name space. The interactions between SN-OSAL and OS are carried out by connecting interfaces at both levels and subsequently, mapping rules between SN-OSAL and OS-specific applications have been identified. Appendix B is the reference bibliography to use in this chapter. It includes the associations between SN-OSAL API and TinyOS 1.x, TinyOS 2.x and Contiki interfaces ($\nearrow_T$).

The process of code generation is performed through the *SN-OSAL Translation Engine*, which includes a pre-compiler of SN-OSAL native code, and the translators between SN-OSAL and each OS. This component performs the *transformation process* between the high-level application (SN-OSAL) into the low-level application (OS-specific). Note that this process is based on MDA standard ideas. Additionally, to completely generate well-formed applications, code adaptations have been accomplished. Finally, the chapter has discussed the correctness of the applications automatically generated by SN-OSAL in terms of its execution model.

Chapter 6

Application layer

In this chapter the highest layer in the architecture proposed is described: the *Application Layer*. The goal at the upper level is to provide users with a comprehensive and simple programming language for writing WSN applications on top of SN-OSAL (SN-OSAL applications). The heterogeneity and complexity at the lower levels is transparent to programmers, who can focus on the specific development details at the application level.

For this purpose, a Domain Specific Language (DSL) to describe generic and portable applications at the *Application Layer* has been created: *Sensor Node* Domain Specific Language (SN-DSL). Both syntax and semantics are introduced. Firstly, the basic rules and resources to write SN-OSAL applications are presented. In particular, program structure, execution model, and the complete name space are reviewed. Limitations and possibilities are also explored. Secondly, a grammar has been created to formally express the *Sensor Node* DSL. This grammar is in *Backus-Naur Form* (BNF), a notation commonly used to recognize programming languages. In this way, syntactic rules that SN-OSAL programs should fulfill are clearly established. In addition to the grammar, *syntax diagrams* are shown in order to graphically represent the grammar derivations.

As a guide for programming, a template for building SN-OSAL applications using the *Sensor Node* DSL is described. As a demonstration, several tests applications are implemented using this template. As will be shown, the process of applications building is more efficient than the traditional one, in terms of development times and ease of programming.

6.1. Programming abstractions at the *Application Layer*

Application Layer can be viewed as the interface between programmers and the sensor node-centric architecture proposed. It must provide the set of abstractions capable of expressing generic and high-level applications on top of the underlying level: the *Operating System Abstraction Layer*. Programmers must use the resources provided by the *Application Layer* while the lower layers are masked by the specific OSAL, as explained in the previous chapter. It means that the particularity of each underlying OS (e.g. execution model, programming paradigm, name space) should remain hidden under these abstractions.

Programming abstractions in the *Application Layer* allow us to define Platform Independent Models (PIMs), in terms of MDA standard. They can be grouped into three categories:

- *Domain-Specific Languages*, which unambiguously specify the syntax and semantics that WSNs applications written in a platform-independent way must fulfill. DSLs must be res-

tricted but must also be expressive and robust.

- *Programming interfaces* provided by the underlying OSAL, specifically, the set of primitives, constructs, and reserved words. Note that the richer these interfaces are, the bigger the set of potential applications to be developed. Obviously, the DSL previously described must include the set of programming abstractions as part of its syntax.
- *Libraries* proceeding from different sources, either exported from the underlying OSAL (such as additional functionality or pre-compiled functions), or libraries from the generic-purpose programming language used by the underlying OS (such as C).

Figure 6.1 represents the generic sensor node-centric architecture and the *Application Layer* components. It shows the four layers of the proposed architecture and the basic components for specifying the *Application Layer* on top of the *Operating System Abstraction Layer*.

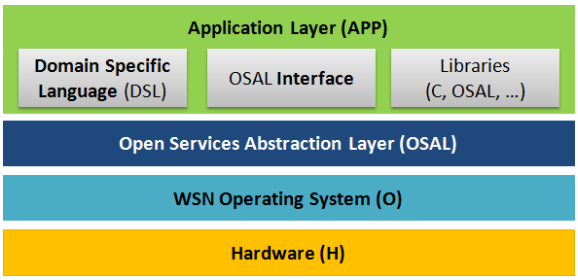

Figure 6.1: Generic sensor node-centric software architecture and *Application Layer* components.

Specifically, SN-OSAL architecture is shown in Figure 6.2. *Application Layer* is located on top of the SN-OSAL, including the *Sensor Node* DSL, the SN-OSAL API and a set of libraries. On the right side of the figure, the *input* is translated to *output* files through a *Transformation Process* carried out by SN-OSAL, as explained in the previous chapter. The following sections describe the programming abstractions described at the *Application Layer* to write applications on top of the SN-OSAL implementation.

6.2. *Sensor Node* Domain Specific Language (SN-DSL)

The motivation for describing a Domain Specific Language on top of the SN-OSAL can be summarized as follows:

- To uncouple the programming language of the underlying OS from the applications programming.
- To hide the execution model exhibited by OSes.
- To define a global and unique space name, and a standardized set of operations that can be used by portable applications.

As described in Chapter 2, TinyOS uses nesC and Contiki uses C for application programming. WSN OSes are not able to hide the execution model exhibited: TinyOS is event- and component-based, while Contiki uses protothreads. Subsequently, given such heterogeneity, in order to write applications on top of them specifying a common programming language is required.

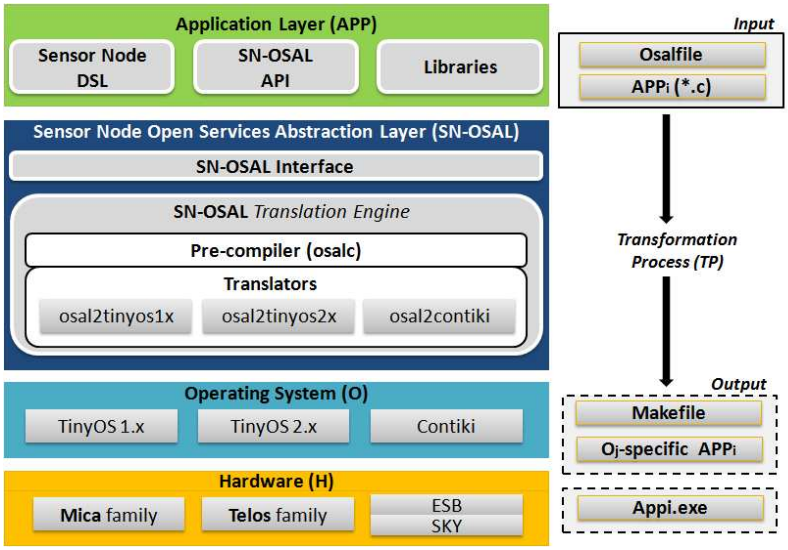

Figure 6.2: The complete SN-OSAL architecture.

Sensor Node Domain Specific Language (SN-DSL) is created to address the development of platform-independent applications, writing on top of the architecture proposed, and using the SN-OSAL presented in the previous chapter. Programmers should use the interface provided by SN-DSL to build applications in a high abstraction level. It allows them to focus on development details because the underlying architecture is presented as a black box. SN-DSL should be able to describe the complete set of potential applications over SN-OSAL, termed $APP_{SN\text{-}OSAL}$.

SN-DSL allows developers to express the WSN applications functionality in a comprehensive way for the *Sensor Node Open Services Abstraction Layer* also, which in a later step must translate them into operating system-specific code.

The next sections are focused on the description of SN-DSL. In particular, its main characteristics, syntax and semantics are presented.

6.2.1. Semantics

SN-DSL uses event-based semantics to build high-level applications. It does not constitute a pure programming language, but it can be viewed as a *metalanguage* because it covers and uses the C programming language, which is the native language used both by nesC and Contiki OS. SN-DSL is a restricted language because it allows to be expressed using a small set of reserved words which map the functionality of sensor nodes and device names. It does not prevent SN-DSL from possibly being further extended to adapt to new requirements. SN-OSAL also uses a declarative syntax which will be explained in subsequent sections. It also includes the SN-OSAL API, explained in the previous chapter, and the libraries as part of its syntax.

Applications on top of SN-OSAL must be written using SN-DSL, and they should be encoded according to the following rules:

- *Event-based semantics* specifies the pieces of code that must be executed when a hardware interruption occurs as the result of a previously invoked high-level service. Additionally, blocking conditions and loops are avoided, as event-model programming is used. Itera-

tions are obtained through timers, which can be configured to expire periodically, and thus synchronizing the application that is working.

- *Declarative syntax* identifies the program structure. The order of the basic blocks of programming into applications must be respected. This order is as follows: event handlers, execution entities, and a unique `main` function. Formally:

$$(event\ handlers)^*(functions)^*(\texttt{main}) \tag{6.1}$$

6.2.1.1. Programming blocks

There are three basic programming blocks that SN-OSAL is able to properly transform into equivalent ones in the underlying OS: `main`, functions, and event handlers. SN-DSL describes the following semantics for each one of them:

- The `main` function constitutes the fundamental piece in an SN-OSAL application, and, subsequently, it is always required. The program statements corresponding to the starting point of execution must be encoded in it. It is equivalent to the homonym `main` function in C programming language.
- *Functions* represent pieces of code which go to execute when the OS scheduler decides. A function is scheduled by `osal_task_create` primitive. Note that the scheduling of these processes will depend on the underlying OS.
- *Event handlers* encapsulate the code to be executed when an interruption is signaled by the hardware. The `osal_task_signal` primitive relates the signal with the event handler to execute. They are analogous to event handlers in event-based models.

A detailed description of these three basic blocks of programming is given below. Furthermore, the order of presentation corresponds to the order of appearance in the SN-OSAL program.

Event handlers An event handler must be encoded for each high-level event. Every high-level event is mapped to a low-level event, as explained in the previous chapter. An event is a hardware interruption such as the completion signal of a previously initiated service or the reception of a message from the network. When an interruption is produced, the associated handler is immediately invoked, and, later the interrupted process is resumed (if the scheduling conditions have not changed). This break of the sequential program flow is performed in a transparent way to programmers.

Nothing in the syntax distinguishes the event handler from a function prototype. However, to specify the former invoking the next primitive is required:

```
void osal_task_signal(int signal, int desc, void* (*eventhandler)(void*))
```

where:

- *signal* is an integer representing the identifier of the event to capture. Name space must register a list of signal identifiers, for instance: `OSAL_IO_READDONE`, `OSAL_IO_WRITEDONE`, `OSAL_TIMER_EVENT`, `OSAL_NET_RECEIVE` or `OSAL_NET_SENDDONE`.

- *desc* acts as a discriminant among the virtual devices affected by the signaled *event*. Additionally, note that the same event could be produced on different physical devices (e.g. sensors, memory), and subsequently, distinct event handlers must be encoded.
- *eventhandler* specifies the function to be executed, or event handler.

Execution entities Execution entities allow a deferred function to be specified, which is scheduled to be executed later in accordance with the scheduling policy of the underlying OS. Specifically, an execution entity could map to a *task* in TinyOS, or to a *protothread* in Contiki. It is important to point out that the details of their execution are platform-dependent.

As event handlers, execution entities are represented as traditional C functions. The prototype is shown below:

```
type functioname (type1 arg1, ..., typen argn) {
      ...
      /* Statements */
      ...
}
```

To initiate an execution entity, the following service should be used:

```
uint8_t osal_task_create(char *name, void* (*routine)(void*), void *args)
```

where:

- *name* is a string representing the identifier of the execution entity to create. It corresponds to the first argument in the function prototype.
- *routine* specifies the starting function to be executed.
- *args* identifies a list of arguments for the execution entity to be created.

This service creates and schedules a virtual execution entity which can be mapped into an equivalent one in the corresponding OS (as mentioned, a *task* in TinyOS or a *protothread* in Contiki). The aspects concerning scheduling and execution are OS-dependent. In this way, SN-DSL abstracts away the details of the execution model of the underlying OS, and at the same time respects their policies.

The set of services related to *Scheduling System* were listed in Table 5.1 from Chapter 5. The library *OS_SchedServices* had to be implemented in order to incorporate these services, which provide a certain high-level control over the execution entities created by the programmer. Applications can use that library, incorporating it into the code through the clause *include*.

`main` function The starting point of a program written on top of SN-OSAL is represented by `main` keyword. As mentioned, this function corresponds to the core of an SN-OSAL program, and subsequently, it is always included in applications. It uses the following template:

```
void main () {
      ...
      /* Statements */
      ...
}
```

The first line to execute will be the first one located in the `main` block. Execution flow is sequential. It is eventually broken by hardware interruptions, which produce events that can be managed by the proper event handlers. Periodic timers drive the application to be executed periodically. From the `main` function, as described previously, different execution entities can also be scheduled.

6.2.2. Syntax

This section focuses on the SN-DSL syntax. SN-DSL syntax includes the set of reserved words by the language that must be used to correctly write applications on top of the proposed architecture. Subsequently, it includes the set of primitives, constructs and name space specified by SN-OSAL (described in the previous chapter). Below, the SN-DSL syntax is described using a grammar developed for language recognition.

6.2.2.1. Name space

The complete name space of SN-DSL is presented in Listing 6.1. It is composed of all reserved words by the DSL. Note that reserved words of C programming language are omitted (e.g. data types or condition structures).

A
ACCEL_X
ACCEL_Y
ADDR_BROADCAST
ADDR_LOCAL
ADDR_SERIALPORT

G
GET_SCALE
GET_INTERVAL
GET_COUNTER

H
HUMIDITY

L
LIGHT
LED_RED
LED_GREEN
LED_YELOW
LEDS_ALL

M
MILLISEC
MICROSEC
MAG_X
MAG_Y
MICROPHONE
main

P
PRESSURE

O
ONE_SAMPLE
OSAL_TIMER_EVENT
OSAL_NET_RECEIVE
OSAL_IO_WRITEDONE
OSAL_IO_READDONE
osal_comm_message
osal_pkt
osal_task
osal_io_close
osal_io_open
osal_io_read
osal_fs_close
osal_fs_delete
osal_fs_lseek
osal_fs_open
osal_fs_read
osal_fs_rename
osal_fs_stat
osal_fs_write
osal_led_on
osal_led_off
osal_led_toggle
osal_net_getId
osal_net_getIdBroadcast
osal_net_getIdSerial
osal_net_send
osal_task_signal
osal_task_create
osal_task_current
osal_task_exit
osal_task_list

```
                        osal_task_pid
R                       osal_timer_ioctl
READ                    osal_timer_restart
REPEAT_SAMPLE           osal_timer_start
                        osal_timer_stop
S                       OS_SchedServices.h
SOUNDER
SEC                     V
SET_INTERVAL            VIBRATION
SET_SCALE
SET_COUNTER             W
                        WRITE
T
TEMP
```

Listing 6.1: SN-DSL name space.

6.2.2.2. SN-DSL grammar

In 1957 Noam Chomsky revolutionized the linguistic field with his book *Syntactic Structures* [Cho57], in which the famous *Chomsky hierarchy* was described to classify formal grammars. Formal grammars constitute a mathematical model to unequivocally express the set of rules able to generate a language. It means that any program written using that language should be recognized by the grammar which generates it. A grammar $\mathcal{G}$ is expressed as a quartet of:

$$\mathcal{G} = \{\mathcal{N}, \mathcal{T}, \mathcal{S}, \mathcal{P}\} \tag{6.2}$$

where:

- $\mathcal{N}$ is the set of *Non Terminal* language symbols, which may produce both other *Non Terminals* and *Terminal* symbols.
- $\mathcal{T}$ is the set of *Terminal* language symbols, which cannot be broken down into any other symbol.
- $\mathcal{S}$ represents the *Initial Symbol*, $\mathcal{S} \in \mathcal{N}$, from which the grammar starts to apply the production rules.
- $\mathcal{P}$ is the set of *Production Rules* that map symbols on the left side of the rule into others on the right side. Each production rule is like: $\mathcal{X} \rightarrow \mathcal{Y}$.

In the Chomsky hierarchy four levels of grammars were classified, where the main difference lies in the rules format of set $\mathcal{P}$. In particular, *type-2* grammars, or *context-free grammars*, comprise those able to generate languages like: $\mathcal{A} \rightarrow \gamma$, where $\mathcal{A}$ is a unique non terminal symbol and γ can be a set of both non terminal and terminal symbols. The languages produced by these grammars are subsequently denominated *context-free languages*. Most programming languages belong to this category, which also present the particularity that they are recognized by a non-deterministic pushdown automaton.

Given that grammars provide a non-ambiguous formalism to recognize languages, this thesis proposes using grammars to formally represent the syntax of the DSLs described over an *Open*

Services Abstraction Layer. This is useful to determine the set of correctly written applications using such DSL. In the particular case of the proposed SN-DSL, it can be formally described by a type-2 grammar, which is composed of the next four sets:

Non terminals ($\mathcal{N}$) *Non Terminal* symbols of grammar are artifices that are intentionally created to produce both *Non Terminal* and *Terminal* symbols. In this way, rules can map other high-level abstractions to encapsulate successive rules. These symbols do not belong to the syntax of the language, but are only intended to generate them.

Terminals ($\mathcal{T}$) The set of *Terminal* symbols represents the alphabet of the language. In this way, it is composed of the reserved words of SN-DSL. Those reserved words proceed from different sources, as mentioned in the previous chapter: constructs, primitives, and the keywords intended to represent settings or argument values. Subsequently, the set of *Terminal* symbols corresponds to the name space shown in Section 6.2.2.1. Additionally, some keywords of C should be included.

Initial symbol ($\mathcal{S}$) The initial string of the grammar ($\mathcal{S}$) corresponds to a special symbol which represents the program to recognize. As mentioned, this symbol must belong to the set of *Non Terminal* symbols.

Production rules ($\mathcal{P}$) Finally, the set of production rules ($\mathcal{P}$) describes the syntax of the language to generate. The process initiates by taking the initial symbol ($\mathcal{S}$), and applying successive production rules until only *Terminal* symbols are obtained, which means that no other rules are available. It is due to the fact that, in type-2 grammars, a *Terminal* symbol cannot appear on the left side of a rule. Applying a production rule means replacing the symbols on the left with those on the right part of the rule. This process is known as *derivation*.

To represent type-2 grammars, John Backus described the *Backus-Naur Form* (BNF) [Bac59]. The BNF notation and its multiple variants have been commonly used to describe programming languages. With this goal in mind, BNF notation is used to express the context-free grammar able to generate the SN-DSL. Therefore, a grammar recognizes the SN-DSL if, and only if, all possible syntactically correct programs written using SN-DSL can be obtained through derivations of this grammar. The notation used in BNF may be summarized in the following sentences:

- *Non Terminal* symbols are enclosed in angle brackets ("<" and ">").
- *Terminal* symbols are enclosed in single quotation marks (""" and """).
- *Production Rules* follow the syntax: $\mathcal{X}$ ::= $\mathcal{Y}$. The symbol "|" is used to express alternatives. The empty string is commonly represented by the keyword "<empty>".

Listing 6.2 presents the proposed BNF grammar able to generate SN-DSL.

```
<snosalapp>    ::=   <main> |
                <function> <main> |
                <header> <snosalapp>.

<header>       ::= <sharp> <include> <quotation> <library> <quotation>.

<library>    ::= 'OS_SchedServices.h' |
                <variable>.

<function>     ::= <functionprototype> <functionbody> |
                <functionprototype> <functionbody> <function>.

<functionprototype>::=  <type> <functionname> <openparenthesis> <arguments> <closeparenthesis>.
```

```
<functionbody>  ::= <openbracket> <body> <closebracket>.

<main>      ::=   <mainprototype> <functionbody>.

<mainprototype>::=  'void' 'main' <openparenthesis> <closeparenthesis>.

<arguments>   ::= <void>
              <type> <variable>  |
              <type> <variable> <comma> <arguments>.

<body>      ::=   <declaration> <statement>.

<statement>   ::= <void> |
              <service> <statement> |
              <condition> <statement>.

<condition>   ::= <initcondition> <service> <closecondition>.

<initcondition>::=  'if' <openparenthesis> <listconditions> <closeparenthesis> <openbracket>.

<listconditions>::=   <simplecond> |
              <simplecond> <operatorcond> <listconditions>.

<simplecond>  ::= <openparenthesis> <variable> <operator> <variable> <closeparenthesis>.

<closecondition>::= <closebracket>.

<declaration> ::=   <void> |
              <type> <variable> <semicolon> <declaration> |
              <type> <asterisk> <variable> <semicolon> <declaration>.

<service>   ::= <sensorservices> <semicolon> <service> |
              <ledservices> <semicolon> <service> |
              <networkservices> <semicolon> <service> |
              <schedulingservices> <semicolon> <service> |
              <storageservices> <semicolon> <service> |
              <timeservices> <semicolon> <service> |
              <otherservices> <semicolon> <service>.

<ledservices> ::=   'osal_led_on' <openparenthesis> <led> <closeparenthesis> |
              'osal_led_off' <openparenthesis> <led> <closeparenthesis> |
              'osal_led_toggle' <openparenthesis> <led> <closeparenthesis>.

<sensorservices> ::=  'osal_io_open' <openparenthesis> <sensor> <closeparenthesis> |
              'osal_io_close' <openparenthesis> <sensor> <closeparenthesis> |
              'osal_io_read' <openparenthesis> <sensor> <closeparenthesis>.

<timeservices> ::=  <variable> <assignation> 'osal_timer_start' <openparenthesis> <value> <comma> <accuracy> <
      comma> <frequency> <closeparenthesis> |
              'osal_timer_stop' <openparenthesis> <variable> <closeparenthesis> |
              'osal_timer_restart' <openparenthesis> <variable> <closeparenthesis> |
              'osal_timer_ioctl' <openparenthesis> <variable> <comma> <cmd> <comma> <value> <closeparenthesis>.

<networkservices> ::='osal_net_send' <openparenthesis> <address> <comma> <variable> <comma> <value>
<closeparenthesis> |
              <variable> <assignation> 'osal_net_getId' <openparenthesis> <closeparenthesis> |
              <variable> <assignation> 'osal_net_getIdBroadcast' <openparenthesis> <closeparenthesis>  |
              <variable> <assignation> 'osal_net_getIdSerial' <openparenthesis> <closeparenthesis>.

<schedulingservices> ::= 'osal_task_create' <openparenthesis> <variable> <comma> <variable> <comma> <variable>
      <closeparenthesis> |
              <variable> <assignation> 'osal_task_current' <openparenthesis> <closeparenthesis> |
              'osal_task_exit' <openparenthesis> <closeparenthesis> |
              <variable> <assignation> 'osal_task_pid' <openparenthesis> <closeparenthesis> |
              'osal_task_list' <openparenthesis> <closeparenthesis> |
              'osal_task_signal' <openparenthesis> <eventname> <comma> <variable> <comma> <variable> <
                   closeparenthesis>.

storageservices ::= <variable> <assignation> 'osal_fs_open' <openparenthesis> <variable> <comma> <mode> <
      closeparenthesis> |
              'osal_fs_close' <openparenthesis> <variable> <closeparenthesis> |
              'osal_fs_write' <openparenthesis> <variable> <comma> <variable> <comma> <value> <closeparenthesis
                   > |
              'osal_fs_read' <openparenthesis> <variable> <comma> <variable> <comma> <value> <closeparenthesis>
                    |
              'osal_fs_rename' <openparenthesis> <variable> <comma> <variable> <closeparenthesis> |
              'osal_fs_lseek' <openparenthesis> <variable> <comma> <variable> <closeparenthesis> |
              'osal_fs_stat' <openparenthesis> <variable> <comma> <variable> <closeparenthesis> |
              'osal_fs_delete' <openparenthesis> <variable> <closeparenthesis>.

<otherservices>::=  <print>.

<print>     ::= 'printf' <openparenthesis> <literal> <comma> <arguments> <closeparenthesis> <semicolon>.

<eventname>   ::= 'OSAL_READDONE' |
              'OSAL_WRITEDONE' |
              'OSAL_RECEIVE' |
              'OSAL_TIMERFIRED'.

<value>     ::= <number> | <variable>.
```

```
<led>       ::= 'RED_LED' |
                'GREEN_LED' |
                'YELLOW_LED' |
                'LEDS_ALL'.

<sensor>    ::= 'TEMP' |
                'HUMIDITY' |
                'LIGHT' |
                'PRESSURE' |
                'VIBRATION' |
                'SOUND' |
                'MICROPHONE' |
                'ACCEL_X' |
                'ACCEL_Y' |
                'MAG_X' |
                'MAG_Y'.

<accuracy>    ::=   'SEC' |
                'MILLISEC' |
                'MICROSEC'.

<frequency>   ::= 'ONE_SAMPLE' |
                'REPEAT_SAMPLE'.

<cmd>       ::=   'SET_INTERVAL' |
                'GET_INTERVAL' |
                'SET_SCALE' |
                'GET_SCALE' |
                'SET_COUNTER' |
                'GET_COUNTER'.

<mode>      ::= 'READ' |
                'WRITE' .

<address>   ::= <hexmark> <hex> <hex> |
                <hexmark> <hex> <digit> |
                <hexmark> <digit> <digit> |
                <hexmark> <digit> <hex> |
                'ADDR_BROADCAST' |
                'ADDR_SERIALPORT'.

<variable>    ::=   <alphabet> |
                <alphabet> <variable>.

<number>    ::=   <digit> |
                <digit> <number>.

<type>      ::=   'void' | 'int' | 'int8_t' | 'uint8_t' | 'int16_t' | 'uint16_t' | 'char' | 'float' | '
     osal_comm_message' | ''osal_pkt' | 'osal_task' .

<include>   ::=   'include'.

<literal>   ::= <quotation> <variable> <quotation>.

<functionname> ::=  <alphabet>.

<alphabet>    ::=   'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | '
     P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
      'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y'
      | 'z'.

<hex>       ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F'.

<hexmark>   ::= '0x'.

<digit>       ::=   '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .

<operator>    ::=   '==' | '>=' | '<=' | '<' | '>'.

<operatorcond>  ::= '&&' | '||'.

<comma>     ::= ','.
<openparenthesis>   ::= '('.
<closeparenthesis>  ::= ')'.
<openbracket> ::= '{'.
<closebracket>  ::= '}'.
<assignation> ::= '='.
<quotation>   ::= '"'.
<semicolon>   ::= ';'.
<asterisk>    ::= '*'.
<sharp>     ::= '#'.
<void>      ::= <empty> .
```

Listing 6.2: BNF grammar for recognizing SN-DSL.

6.2.2.3. Syntax diagrams

Type-2 grammars can be graphically expressed through *syntax diagrams* [Con63], also known as *railroad diagrams*, which constitute an effective and simple way of expressing each production rule, allowing to clearly show mechanisms such as sequence, alternation, and recursion. A syntax diagram is a special type of directed graph, in which nodes represent both *Non Terminal* and *Terminal* symbols, and arcs represent the sequences in which these symbols can appear in accordance with the grammar. *Terminal* symbols are represented through rounded-boxes and *Non Terminal* symbols are depicted as squares.

Syntax diagrams of the previously presented BNF grammar have been generated using the tool available online in [JB07]. Figures 6.3, 6.4, 6.5, 6.6 and 6.7 show the main *production rules* of the grammar presented in Listing 6.2. To understand the syntax diagram, consider for instance the first graph in Figure 6.3. It represents the productions obtained from the *initial string* $\mathcal{S}$:

```
<snosalapp> ::= <main> |
                <function> <main> |
                <header> <snosalapp>.
```

As depicted, each alternative is drawn as a branch proceeding from the entry (left symbol), where each branch contains different symbols. In this way, three branches have been generated from <*snosalapp*> initial symbol: the <*main*> symbol representing the most simple case with a single `main` function; the <*function*> and <*main*> symbols represent a set of one to *n* functions, which may be event handlers, followed by the `main` function. Finally, the last branch takes into account the apparition of a list of header files (through the C preprocessor directive `include`) or variable declarations. File headers must precede all functions in the program. Note that the <*main*> symbol is always required. The interpretation is similar for all rules shown.

Figure 6.3 shows the *production rules*: from the symbol <*snosalapp*> to <*initcondition*>.

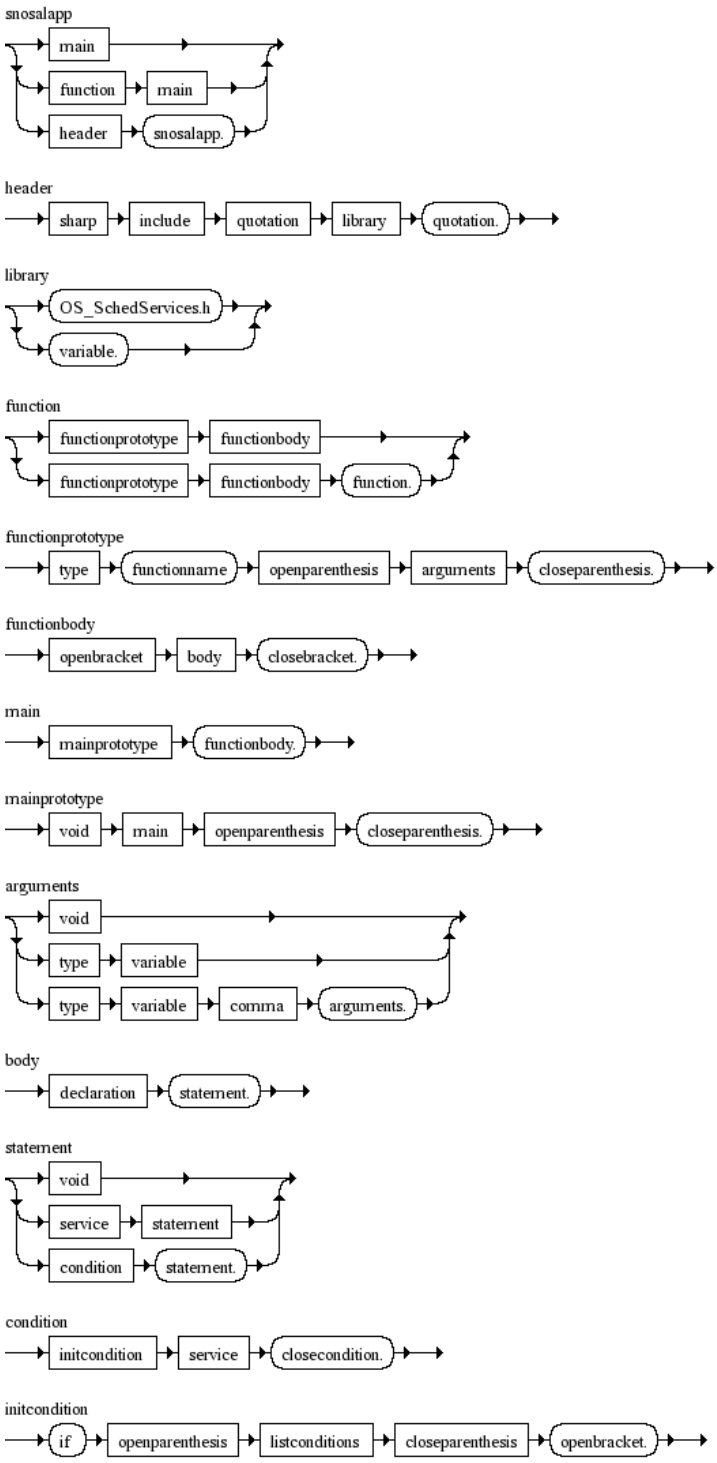

Figure 6.3: SN-DSL grammar: Syntax diagram (Image 1).

Figure 6.4 shows the subsequent *production rules*: from the symbol <*listconditions*> to <*timeservices*>.

Figure 6.4: SN-DSL grammar: Syntax diagram (Image 2).

Figure 6.5 shows the subsequent *production rules*: from the symbol <*networkservices*> to <*eventname*>.

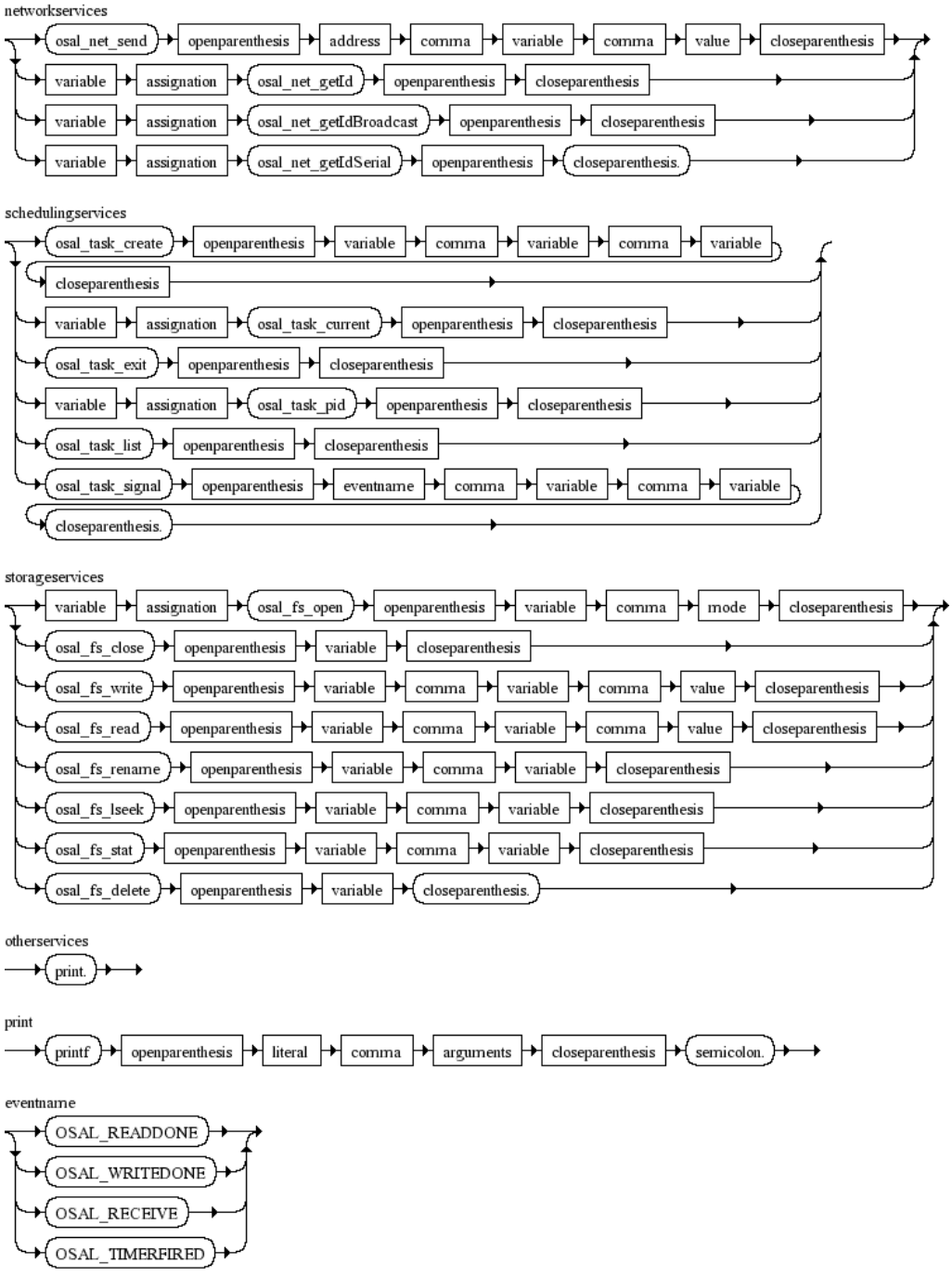

Figure 6.5: SN-DSL grammar: Syntax diagram (Image 3).

Figure 6.6 shows the subsequent *production rules*: from the symbol <*value*> to <*cmd*>.

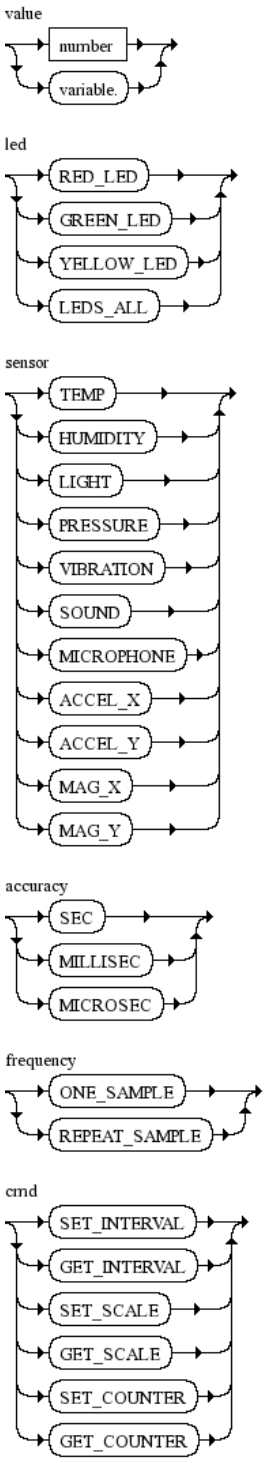

Figure 6.6: SN-DSL grammar: Syntax diagram (Image 4).

Finally, Figure 6.7 shows the subsequent *production rules*: from the symbol <*mode*> to <*literal*>. Note that some *production rules* of type $\mathcal{N} ::= \mathcal{T}$ have been omitted because their representation is trivial. For example the <*alphabet*> symbol would produce all alphabet letters, both upper and lower case, and therefore its associated syntax diagram is evident.

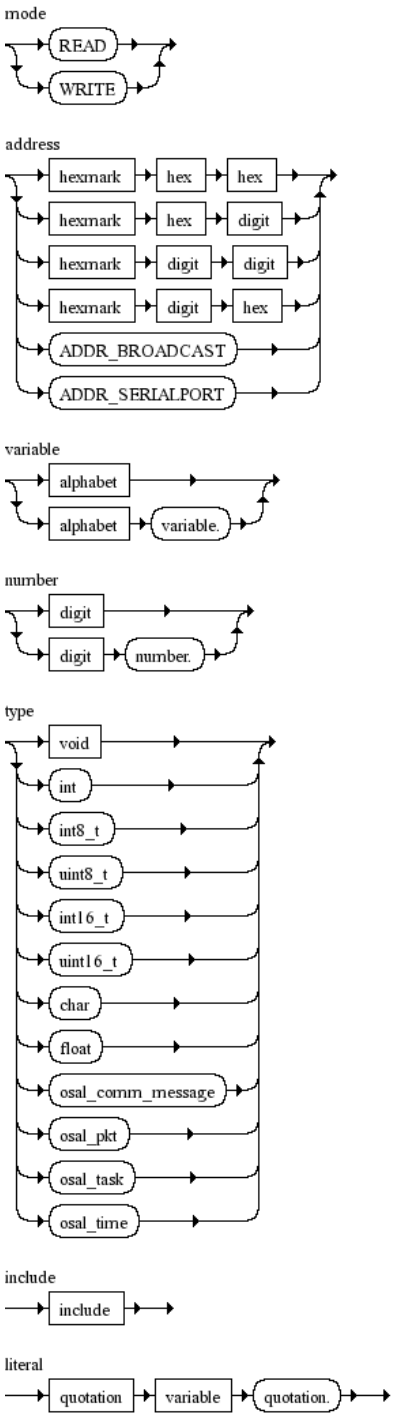

Figure 6.7: SN-DSL grammar: Syntax diagram (Image 5).

6.3. Libraries

Some libraries developed in C programming language can be included into SN-DSL programs. In the current prototype, the functions corresponding to the management of *Tasks & Scheduling System* were developed into the *OS_SchedServices.h* library. In addition, functions related to management of arrays of characters (string) were collected into a library.

6.4. Applications programming on top of SN-OSAL

This section details the applications building on top of SN-OSAL. The set of applications that can be written on top of SN-OSAL was denominated $APP_{SN\text{-}OSAL}$, where each application $APP_{SN\text{-}OSAL_i} \in APP_{SN\text{-}OSAL}$.

Applications must be programmed using SN-DSL. The process of applications building basically consists of writing two files which are provided to SN-OSAL (see Figure 6.2):

- The *application file* containing the implementation itself. It is a text file with *.c* extension, including SN-DSL code in accordance with previously described writing rules.
- The *settings file* containing the platform specifications for which SN-OSAL will generate specific code: the *Osalfile* file. The previous chapter described the format of this file (see Section 5.4).

Both files are required by the underlying OSAL, in order to start the process of code generation. Previously, the pre-compilation step of the application file must be performed. If `osalc` precompiler detects some kind of error, the process is interrupted and the control is returned to the user at the application level. If there are no errors, it proceeds to invoke the suitable translator for code generation. Figure 6.8 shows this process.

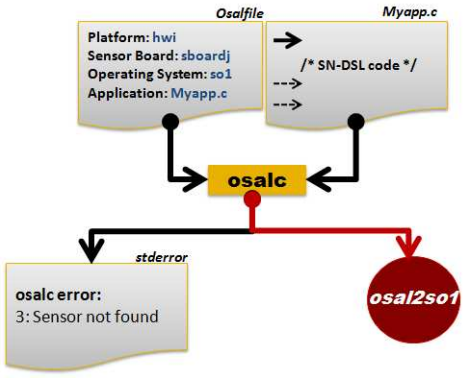

Figure 6.8: Pre-compilation process of an SN-OSAL application.

6.4.1. SN-OSAL program template

To support this process, an application template is given, from which different test programs will be written. Listing 6.3 shows a template for applications that use SN-DSL. It shows the basic components of an SN-OSAL program: events, execution entities, and `main` function. Their order must be respected.

```
/* ----------------- Event handlers -------------------- */
void eventname(int8_t descriptor) {
      /* Statements */
}
/* ----------------- Execution entities ----------------- */
return-type functionname(type1 arg1, ..., typen argn) {
      /* Statements */
}
/* ----------------- main block ------------------------- */
void main() {
     /* The main code */
}
```

Listing 6.3: SN-OSAL program template.

The event handlers are located in the beginning of the application file. A set of zero to *n* event handlers can appear in the file, but programmers are forced to implement one event handler per signal to manage. Events map hardware interruptions and, normally correspond to asynchronous responses to actions that have been previously initiated at any part of the application code, such as completion of a previously initiated reading or writing, network reception event, and so on.

Next, the implementation of the execution entities takes place. They are optional mechanisms provided by SN-DSL to implement a certain functionality which is scheduled to be executed later, depending on the scheduling policy of the underlying OS. The decision to create new execution entities is taken by programmers. It allows a long and complex piece of code to be broken down into simpler functions, favoring the programs modularity. As explained, these functionalities map to *tasks* in the case of TinyOS, or *protothreads* in the case of Contiki OS.

Finally, the mandatory function is provided: the `main` function. Programmers must always implement it. This block includes the statements of SN-DSL code that correspond to the application behavior.

6.4.2. Examples

Taking the previous template, two simple SN-OSAL applications have been developed as an example:

- Periodic sending of data to network.
- Sampling a temperature sensor.

Listing 6.4 shows the first. The main function starts a periodic timer every five seconds and writes an event handler to manage timer expiration. Note that the connection between the event and the associated event handler is made through the `osal_task_signal`. In this event handler, a simple network message is composed. Note that programmers must customize the structure to be sent on the network: the `osal_comm_message`, whose definition is included in the header file *"osal_message.h"*. The program is kept in an infinite loop because the timer was defined as periodic through `REPEAT_SAMPLE` option.

```
#include "osal_message.h"

osal_comm_message osal_pkt;
```

```
uint8_t desc;

void timer(int8_t desc){
  osal_pkt.number = 253;
  osal_pkt.id_nodo = osal_net_getId();
  osal_net_send(ADDR_BROADCAST, &osal_pkt, sizeof(osal_pkt));
}
void main(){
  desc=osal_timer_start(5, SEC, REPEAT_SAMPLE);
  osal_task_signal(OSAL_TIMER_EVENT, desc, (void*)&timer);
}
```

Listing 6.4: SN-OSAL application for periodic sending of data.

Listing 6.5 shows the second example. In this example two execution entities are created from the `main` function. Note that the library *OS_SchedServices* must be included because some services related to *Tasks & Scheduling System* constitute an additional contribution (such as identification of tasks through a `tid` or list of tasks scheduled), and they do not find a mapping function into the OSes. `osal_task_create` schedules a new execution entity in the queue of tasks. The decision of when the task executes is taken by the OS scheduler. In this way, a list of scheduled tasks is managed by this library, in order to include knowledge about the tasks created. If a task finishes (through `osal_task_exit`, or simply ending the function), then it is removed from the list.

```
#include "OS_SchedServices.h"
void helloworld1() {
  osal_task *q = osal_task_current();
  printf("Exec::: Hello World in execution entity : \%d\n", q->
      osal_tid);
  osal_task_exit();
}
void helloworld2(){
  osal_task *r = osal_task_current();
  printf("Exec::: Hello World in execution entity : \%d\n", r->
      osal_tid);
  osal_task_exit();
}
void main(){
  osal_task *p;
  p = osal_task_current();
  printf( Exec::: Hello World in main function : \%d\n, p->
      osal_tid);
  osal_task_create("helloworld1",(void*)&helloworld1, null);
  osal_task_list();
  osal_task_create("helloworld2",(void*)&helloworld2, null);
  osal_task_list();
}
```

Listing 6.5: SN-OSAL application with several execution entities.

Each execution entity is represented by an `osal_task` structure, which contains useful information about the task: an identifier (tid), a pointer to the function to implement, the list of ar-

guments, etc. The length of the structure is about 63 bytes per task, which means that the massive usage of processes could exhaust the RAM memory. Therefore, the programmer is responsible for the correct usage of this library.

6.5. Chapter summary

This chapter has dealt with the development of WSN applications using the architecture described in this thesis. It discusses the highest abstraction level, the *Application Layer* and subsequently, it is intended to provide programming abstractions for developing applications in a simple way, but at the same time without forfeiting to the control that programmers should have over applications. This issue has been handled in two ways:

- First, describing Sensor Node Domain Specific Language (SN-DSL). It is intended to provide the necessary abstractions to write generic applications on top of the lower layer, the *Sensor Node Open Services Abstraction Layer*.
- Second, presenting the type-2 grammar, using the BNF notation, able to generate the SN-DSL.

SN-DSL is also a metalanguage of C programming language, adding a small and global name space able to represent most device names or other reserved words. The basic blocks of building applications have been presented. The writing rules have been explained in order to state the route that applications should take. The BNF grammar associated with the previously described language has also been elaborated. The advantage of using a grammar is that it eliminates the ambiguity related to the programming syntax. Additionally, syntax diagrams corresponding to the main production rules of the grammar were depicted.

Subsequently, a formal method for writing platform-independent applications has been proposed. Applications are portable because they do not take into account the underlying OS execution model, and they can be automatically transformed into the equivalent ones, thanks to SN-OSAL, which carries out the *transformation process* according to MDA standard.

The advantages can be summarized as ease of development, savings of time and resources, and robust and error-free programming. In spite of this, the process could still be substantially improved through a tool for composing applications in a graphic and more intuitive way. The next chapter focuses on this concern.

Chapter 7

Graphical development framework: VisualOSAL

The previous chapter dealt with the problem of describing programming abstractions at the highest layer of the proposed architecture, the *Application Layer*. As a consequence, portable applications can be written on top of the *Operating System Abstraction Layer* (specifically SN-OSAL), which is intended to accomplish the transformation process that generates the equivalent OS-specific applications.

The abstractions that have been defined allow text-based programming. By hand, programmers must create and edit the text files where the implementation will reside. Programming task is error-prone by nature: in addition to the inherent complexity of writing WSN applications, programmers may enter typos, declare the constructs incorrectly or in an equivocal order, introduce variables which will not be used, etc.

This chapter introduces a graphical development framework for composing applications on top of SN-OSAL, which is called *VisualOSAL*. *VisualOSAL* allows WSN applications to describe based on the DSL defined, but where the language elements (e.g. primitives, programming blocks) are substituted with a graphical notation. The benefit of using a visual tool is twofold: 1) to guide the development process through an *Integrated Development Environment* (IDE) in order to avoid programming errors, and 2) to facilitate high-level programming through a graphical interface. It will result in shorter development time and, less effort, and consequently, in an increase of productivity.

7.1. Overview

Visual programming allows programs to be specified using graphical notation instead of textual representation. Tools created for this purpose (e.g. LabView, Ptolemy, SCADE) increase the knowledge that multidisciplinary groups have of a system, facilitate the programming (in general, users prefer pictures to words), bringing benefits in terms of development effort and time, and thus increasing productivity.

The key idea is that a graphical development environment can be developed with the aim of making the programming on top of the proposed architecture easier. This tool should provide a graphical notation for each textual element specified by the underlying DSL, and allow the program sequence to be described according to the syntax and semantics specified. Finally, this

tool will act as a translator between graphical and textual syntax.

Specifically, over the previously described *Sensor Node Open Services Abstraction Layer*, a tool has been created: *VisualOSAL*. *VisualOSAL* addresses the following goals:

- Composition of generic WSN applications through a graphical representation of the elements specified by *Sensor Node Domain Specific Language* (SN-DSL). This DSL was described to program applications over SN-OSAL. As mentioned, graphical building of applications is desirable in order to ease the programming.
- Support for the complete transformation process: from the visual application to the equivalent binary code (see Figure 7.1), including code verification at each of the translation levels.
- Code generation to different WSN operating systems and hardware platforms, and deployment of applications over the previously specified platform.

VisualOSAL supports the complete process of design, implementation and deployment of WSN applications using the proposed architecture. The process consists of the following steps:

- Graphical composition and design of generic WSN applications. For this purpose, a notation graphic is used, as will be shown in subsequent sections. Additionally, the platform settings (e.g. OS, sensor node, sensor board) for which the application will be generated must be specified.
- Successive code translations and generation (from PIM to obtain the equivalent PSM):
  - From the graphical application, to obtain its textual representation as the equivalent SN-OSAL application, which is described using the SN-DSL specified.
  - From the SN-OSAL application, generating the equivalent OS-specific code, using the translation process described in Chapter 5.
  - The compilation environment must also be generated, according to the previously specified user requirements.
- Compiling and linking the application.
- Deployment of applications. The generated application is downloaded into the specific microcontroller.

Figure 7.1 shows the translation process of applications, from the highest-level specification using a graphical tool until the equivalent executable code is obtained. Applications can be described using VisualOSAL, which is able to generate specific code for the underlying OSAL, specifically SN-OSAL, and, finally it generates the equivalent OS-specific code.

7.1.1. *VisualOSAL* features

VisualOSAL allows a program to be specified as a sequence of graphically represented actions, connected by arrows. Each of these actions corresponds to a programming element (e.g. primitive, construct, control structure) specified by the underlying programming language (SN-DSL). Arrows indicate the direction of the sequence. For each graphical element a set of properties is considered, according to its specification. For example, for primitives it should be possible to

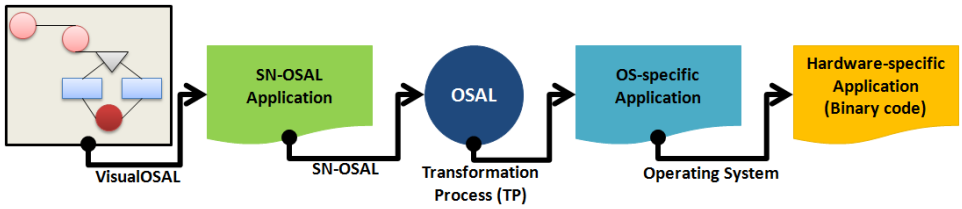

Figure 7.1: Flow of code transformations: from a high-level application (PIM) to a specific one (PSM).

configure in development time the value of parameters included in their prototype. The sequence in which these elements can be arranged is verified using the grammar shown in the previous chapter. According to this grammar, it would be possible to specify a set of header files and a set of global variables. *VisualOSAL* also presents the following features:

- Graphical composition of generic and portable WSN applications through a graphical notation that replaces the textual elements: primitives, programming blocks, and other language elements such as conditional structures.
- Applications are created using *VisualOSAL* projects, whose configuration is written in an XML file. This XML file contains the metadata and description of application contents. In this way, it can be used to save and edit VisualOSAL projects in a common, platform-independent format.
- *VisualOSAL* allows code templates to be added to the project. These templates, which are called *user defined functions* can be customized by the developers, and they are used to introduce source code or, eventually, indicate the path where the function is located, giving greater flexibility to developers
- Multiplatform. Support for different WSN operating systems (TinyOS 1.x and 2.x, and Contiki), and a set of hardware platforms (Mica and Telos family, ESB, SKY).
- Code verification through XML Manifests that were developed for hardware components (see Chapter 4). As explained, XML Manifest contains the characteristics, interfaces, and compatibilities of a particular component. In this way, XML Manifests can be used to check for errors at the implementation time.

7.1.2. *VisualOSAL* applications

One visual application $VAPP_j \in VAPP$ using VisualOSAL can be viewed as a composition of graphical elements g_i connected by arrows. In this way, it can be expressed as:

$$VAPP_j = \{g_0, g_1, g_2 \dots g_n\} \tag{7.1}$$

where g_i is connected to g_{i-1} and g_{i+1}. They can belong to one of the following groups:

- *Primitives* (see Table 5.1). This is the set of primitives exported by SN-OSAL, representing the set of standardized services to compose WSN applications at a high-abstraction level. Every primitive must be configured according to its prototype.

- *User defined functions* are intended to encapsulate code. They can represent two programming blocks for applications building according to SN-DSL: event handlers and functions. In this way, a *user defined function* can be marked with the property "handler" to indicate an event handler. Note that the *main* function corresponds to the actions sequence described graphically.

- *Connectors* represent the conditional structures used by SN-DSL to make decisions about the flow of the program (`if` and `switch`) based on the evaluation of a determined condition.

- *Sentences* allow single sentences to be written using SN-DSL inside a box, such as data initialization or print debug messages, among others.

7.2. Graphical notation

For every group previously mentioned, *VisualOSAL* uses a graphical notation to represent it, in order to carry out a visual composition of applications. This process consists of selecting the suitable icons identifying the desired function, and arranging them on a drawing panel, which contains the visual representation of the application. The following sections describe the notation used.

7.2.1. Primitives

Every functionality considered is represented by an intuitive graphical symbol, such as icons, which allow the set of services in each group of functionality to be expressed. Figure 7.2 presents the set of icons for graphical representation of the functionalities taken into account for WSN programming, from left to right: *Leds, File system, Timers, Network & Comm, I/O, and Tasks & scheduling*. Thus, for any primitive in *Leds* functionality, the first icon must be selected and dropped onto the drawing panel. Note that the same symbol is used to express the whole set of services described for each functionality.

Figure 7.2: Graphical notation for representing the six functionalities (from left to right): leds, file system, timers, networks, I/O, tasks & scheduling.

Developers must select and configure the action required when the icon is selected. For this purpose, next to the icon, a dialog appears to enter the configuration of the primitive. Firstly, from a *Combo Box*, users must select the service from the set of available primitives in each group. Secondly, once the primitive to configure is chosen, they must set the values for the arguments of the function.

7.2.2. User defined functions

Alternatively, users can use the textual notation for adding actions during applications development. For example, developers can encode an event handler or function by hand. It can be useful if the code is already available. To achieve this, *User defined functions* are intended to encapsulate within a set of sentences. The language to use inside these boxes is SN-DSL, the *Domain Specific Language* specified for applications building over SN-OSAL.

Figure 7.3 shows the graphical notation for *User defined function*. As shown, it is represented by a box in which the user can enter the text corresponding to program sentences. Every *User defined function* must have a unique name.

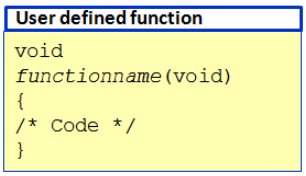

Figure 7.3: Graphical notation for a *User defined function*.

7.2.3. Sentences

Finally, to represent other valid sentences in a program, a similar notation to *User defined functions* is used. In particular, three types of sentences are considered: the statement `include` for header files, *data section* for declaring global variables, and finally any other statements not included in the previous description, such as data assignation to variables, debugging sentences or even comments. Figure 7.4 shows the graphical representation for *data section*, *includes* and other sentences. Note that the symbol "+" at the bottom means that several instances can be added.

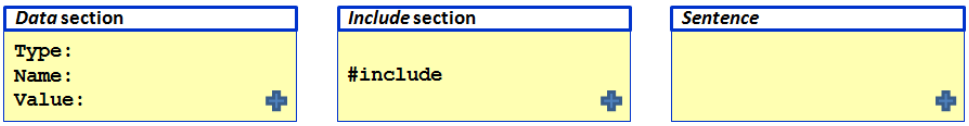

Figure 7.4: Graphical notation for *data section*, *includes*, and other sentences.

7.2.4. Connectors

There are two possible connectors between symbols: arrows and conditions. *Arrows* allow the flow of execution of the represented actions to be specified. Note that they do not indicate the temporal order of the actions since events can preempt the task in execution. *Conditions* express the evaluation of a program element, such as a variable or the occurrence of an event. The former provides the conditional control structure (e.g. if, switch), while the latter allows handlers to be

specified after the event occurs. Figure 7.5 shows the notation employed for arrows and conditions.

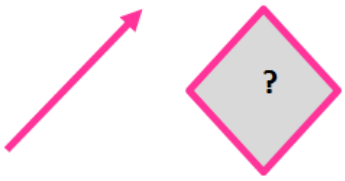

Figure 7.5: Graphical notation for connectors: *arrows* and *conditions*.

As explained in the previous chapter, loops are restricted according to event-based programming: a loop can prevent event reception while it is being executed.

7.3. Visual programming using *VisualOSAL*

VisualOSAL is an IDE for graphical composition of WSN applications. It allows the application requirements and the target environment to be specified completely. Application requirements are related to the target platform (e.g. mote, sensorboard, gateway) and the application itself (e.g. take a sample of temperature every n seconds, send data, write data to the flash memory).

Through a graphical user interface the complete set of requirements can be introduced. The framework proposed provides a central panel where the graphical design of the application is made, by selecting icons and dropping them onto the drawing panel to compose the application. Additionally, the central panel encloses the textual representation in different tabs using SN-DSL and, when generated, using the operating system programming language. On the left, the project properties are broken down: file headers, variables and user defined functions. Figure 7.6 shows the appearance of *VisualOSAL* where the previously described elements can be identified.

VisualOSAL gives support to the entire life cycle of applications development using the architecture proposed in this thesis. In particular, it covers from the application design to the deployment of the network. The phases of this life cycle can be summarized as:

- *Phase 1: Design* consists of the graphical composition of the application using the graphical notation provided.
- *Phase 2: Application generation* accomplishes code generation in two subprocesses: firstly, from the SN-DSL code into the equivalent OS-specific code, and secondly, compiling the latter to obtain the executable code.
- *Phase 3: Deployment* installs the binary code into the microcontroller of the platform specified in the first phase.

The *output console* and *error console* located at the bottom of the framework are intended to show the results obtained in each phase. The following subsections describe each development phase in depth.

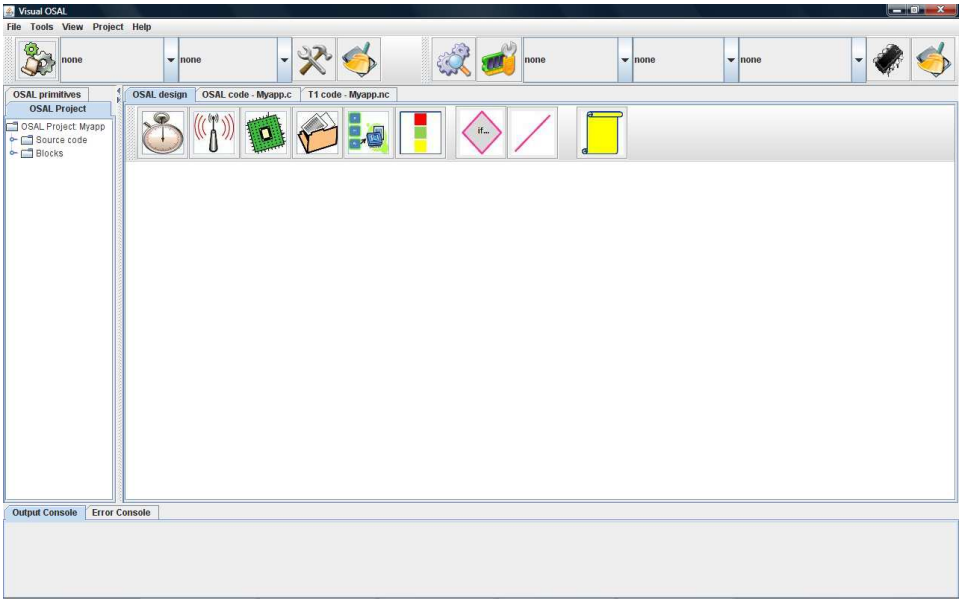

Figure 7.6: *VisualOSAL*: a graphical development framework for composing applications using SN-OSAL.

7.3.1. Graphical design

The first phase is intended to develop the WSN application using the graphical notation provided by *VisualOSAL*, which consists of the set of elements described in Section 7.2. By default, developers start building the `main` function, defining the set of actions, event handlers, user defined functions and other elements required by the application. As described, every action is configured at development time, selecting the primitive name and its arguments.

Users must select the target platform where the binary code will be installed, and the settings for the programming board to download code. Note that this process is equivalent to writing the *Osalfile* file, which contained the details of the target platform.

The *VisualOSAL* framework uses XML Manifests in order to describe user requirements. Consider the GUI presented in Figure 7.7. User requirements are provided through graphical elements such as menus, combo boxes or buttons. For instance, a menu containing the sensor boards models is displayed when the user tries to choose the specific sensor board to be programmed. In this way, the menu shows the logical name of every sensor board (for which a XML Manifest must exist). XML Manifests were introduced in Chapter 4.

Figure 7.7: Menu for settings description in *VisualOSAL*.

Composing visual applications avoids the introduction of syntax errors (as the code is generated), while the manual writing of applications does not. Nevertheless, semantics errors can be

detected in this initial phase of development due to the use of the XML Manifests. For example, if a sensor board is selected, the set of available sensors is known, and developers can only choose from this set.

Figure 7.8 shows the *Blink* application depicted on the drawing panel. As shown, the settings of the platform where the application will be installed are specified: TinyOS 2.x and Telos, as well as the options for programming: /dev/ttyUSB0 port.

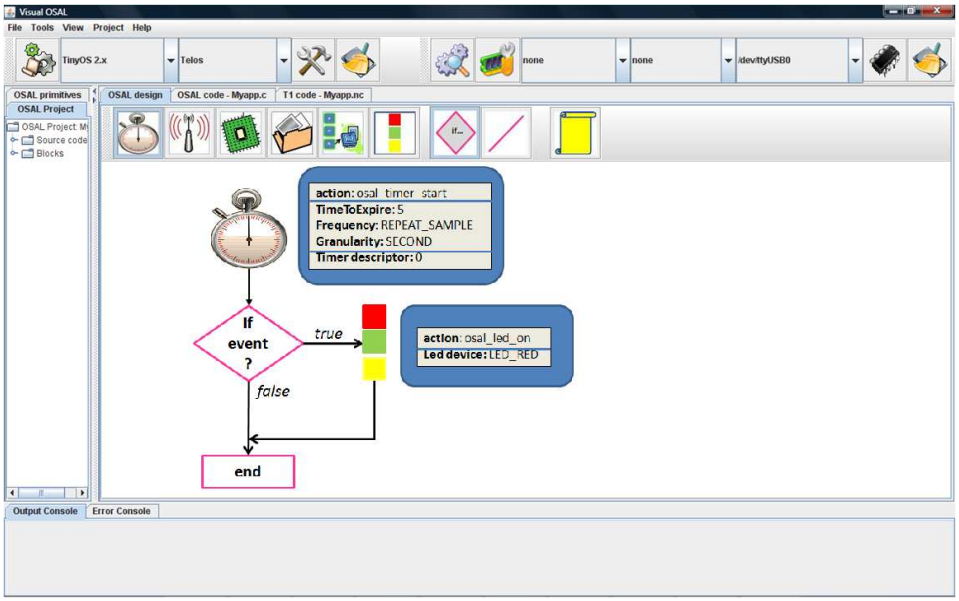

Figure 7.8: Generic *Blink* application designed using *VisualOSAL*.

Note that the graphical representation of WSN applications through *VisualOSAL* is closer to MDA standard ideas, which promotes the use of graphical models (e.g. UML) to specify its Platform Independents Models (PIMs) and Platform Specific Models (PSMs).

7.3.1.1. Portable representation

Design should be able to be stored to be further manipulated. A *VisualOSAL* project is a complete specification of the contents, properties and values of each project at a moment of the development phase. Thus, this specification must allow different projects can be identified and managed.

The representation to be used must be portable to facilitate integration. For this reason, XML is the format chosen for describing *VisualOSAL* projects. Subsequently, it can be used to interchange and modify the project design between different working groups. The XML file is dynamically updated at development time, for example when developers add a new action on the drawing panel or modify the project configuration. As an example, consider the Listing 7.1, which contains the XML file for describing the Blink application using the *VisualOSAL* framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE visualosal SYSTEM "VisualOSAL.dtd">
<Project>
  <Manifest signature="vosalproject-0000000001">
```

```
    <definition>
      <name>VisualBlink</name>
      <path>c:/VisualOsal/workspace/MyVisualOsalProject/</path>
      <src>./snosal</src>
      <bin>./bin</bin>
      <doc>./doc</doc>
    </definition>
    <target>
      <operatingsystem  name="TinyOS" version="2.x" compiler="
          nesc" path="/opt/tinyos-2.x"/>
        <platform>
          <mote   name="Telos"  provider="Crossbow Technology"
              signature="/platforms/Telos/Telos-00000001.xml"/>
          <sensorboard></sensorboard>
        </platform>
    </target>
    <deployment>
      <port   name="/dev/ttyUSB0"/    type="USB"  OS="Linux"/>
    </deployment>
    <application>
      <include></include>
      <data></data>
      <text>
        <action id="001"  name="osal_timer_start" functionality=
            "timer" ReturnValueType="int8_t">
          <param name="timeToExpire"  type="int8_t" value="5"/>
          <param name="frequency" type="int8_t" value="REPEAT"/>
          <param name="granularity" type="int8_t" value="SEC"/>
        </action>
        <condition>
          <if operand="event" operator="equal"  operand="timer">
            <action id="002"  name="osal_led_on" functionality="
                leds" ReturnValueType="void">
              <param name="led" type="int8_t" value="LED_RED"/>
            </action>
          <else></else>
          </if>
        </condition>
      </text>
    </Manifest>
</Project>
```

Listing 7.1: XML Manifest describing a *VisualOSAL* project.

Note that the XML file describing a *VisualOSAL* project is bidirectional: it allows both the textual representation of the design expressed as a SN-DSL application, and its graphical representation to be obtained with no ambiguity.

7.3.2. Application generation

This phase consists of translating code between two textual representations of application: from the high-level application in SN-DSL to the equivalent low-level representation in binary code. In other words, it is intended to generate platform-specific application code from its generic

form. Note that the SN-DSL code is obtained from the portable representation explained previously. Figure 7.1 shown this process, which can be divided into two sub-processes:

1.- Firstly, from SN-DSL to OS-specific programming language, using the translation process ($\mathcal{TP}$) described in Chapter 5.

2.- Secondly, from OS-specific code to binary code through the platform-dependent compiler. The compilation environment is also obtained from the application properties.

In order to generate code, developers must select the *Code generation* option from the *Tools* menu. This process is automatic and transparent to developers. If errors are found, the *error console* located on the bottom will show the error and its corresponding explanation. Otherwise, certain output is produced in this phase: source code files obtained in both subprocesses (SN-DSL and OS-specific code), and the binary code to be installed on the target sensor node. *VisualOSAL* framework allows the source code files generated in the project to be visualized in different tabs.

7.3.3. Deployment

The last phase of the life cycle consists of downloading the executable code obtained from the previous phase into the microcontroller of the sensor node that has been selected. This process must be repeated for every sensor node within the WSN. To access this option, developers must connect the serial interface and the programming board to the proper port and, select the *Deployment* option from the *Tools* menu in order to carry out the installation.

7.4. Chapter summary

Due to the fact that the proposed architecture has been formalized in different abstraction layers and the API used by the applications built on top is clearly specified, a tool for visual programming could facilitate applications development as well as to increase the productivity, and, consequently, extending the WSN technology use to non-expert groups.

This chapter has described a graphical development framework to compose generic WSN applications over the architecture proposed in this thesis, *VisualOSAL*. Besides providing an IDE, it also gives a semi-automatic support for the life cycle of WSN applications, using the architecture proposed. In particular, three phases have been identified: design, application generation, and deployment. Each of these phases has been described in this chapter, identifying how *VisualOSAL* can help developers. As explained, certain output is produced: high-level code (in SN-DSL), operating system-specific source files (as an intermediate step), and the final executable code ready to install into the target sensor node. While there is a unique visual representation of an application, *VisualOSAL* is able to transform it into the equivalent application in different WSN operating systems, therefore providing portability.

Since the framework is at an advanced state of development (but not yet finished), this chapter has focused on the main guidelines to implement a tool which assumes the underlying architecture described. Specifically, it is located over the *Sensor Node Open Services Abstraction Layer* (SN-OSAL), described in Chapter 5. Therefore, it uses a graphical notation to express the SN-DSL elements in order to compose visual applications.

A preliminary study published in [ECIT08] established the basis of development for the *VisualOSAL* framework.

Chapter 8

Evaluation

This chapter describes the evaluation of the proposed architecture. The goal is to determine the portability level achieved by high-level applications, quantify the footprint overhead due to the SN-OSAL usage, and finally, estimate the cost of applications development. These aspects provide a measurement of the feasibility of employing the proposed architecture and its contribution to WSN usage.

In order to accomplish this, several prerequisites were established: firstly, the metrics and the target platforms (operating system and hardware) considered; and secondly, the benchmark applications. Experiments consisted of programming every benchmark from scratch, using each of the OSes and using SN-OSAL. Through this process both the applications written using SN-OSAL API, and the targets were obtained. Measurements of the different metrics were obtained and compared. The experimental results are presented in this chapter.

In addition to considering the portability and the footprint, software cost estimation has also been quantified. The COCOMO II model will be used to carry out the estimation of effort, development time and productivity of the applications built using SN-OSAL, and the applications automatically generated for the supported operating systems. This model will help to determine whether the hypothesis of this thesis, stated in Chapter 3, has been proven.

As a demonstration, a real-world application has been developed using SN-OSAL. The target code generated from it is described and then measured in terms of portability, footprint, and productivity.

As shown, SN-OSAL applications are portable to heterogeneous sensor nodes both in hardware and operating system. The overhead introduced by SN-OSAL is not meaningful, as it meets the critical design goal of minimal redundancy established in previous chapters.

8.1. Evaluation goals

WSN applications evaluation is a task facing new challenges. The standard metrics used to offer a measurement of the application performance in conventional environments, such as execution time, are not always valid in this scope. On the contrary, there are frequently other less critical metrics in other environments, such as energy or applications footprint, which reveal the degree of efficiency of WSN applications. Related to the architecture evaluation, three were the goals posed:

- Determining the degree of *portability* achieved by SN-OSAL applications, in other words,

estimating the range of applications and platforms which SN-OSAL could be used for.

- Evaluating the process of WSN applications programming using SN-OSAL, in terms of *ease of programming* and design, and *productivity*.

- Quantifying applications *efficiency*, that is, determining the overhead imposed by SN-OSAL over the original applications. Performance is usually measured by the footprint and the executable code size due to the severe constraints of motes.

8.2. Metrics

The previous evaluation goals have conditioned the metrics to be considered. To determine efficiency, a static analysis based on applications footprint will be carried out. To estimate the degree of portability of SN-OSAL applications, an analysis of the number of WSN operating systems and motes for which SN-OSAL is able to generate code will be taken into account. Finally, to estimate the software costs and productivity, the COCOMO model will be applied. In this section the description of metrics is presented.

8.2.1. Metrics for applications analysis

A static analysis allows the measurements of efficiency associated with applications to be determined, in order to clearly identify when an application is more efficient than another. This analysis is performed at compilation time and subsequently, there are two observable measurements: applications *footprint* and executable code size.

The previous metrics are computed at compilation time, and they provide an idea of the efficiency in the applications development. The following subsections summarize the process for obtaining these values. To get a measurement of portability, other indicators must be taken into account, such as the number of target platforms for which the code could automatically be generated through SN-OSAL, and the degree of accuracy between the original and the target application.

8.2.1.1. Footprint and executable size

Footprint is defined as the amount of space occupied by a piece of software. As mentioned, especially in WSN, the footprint is particularly important due to hardware limitations. The footprint is usually set by the RAM and ROM memory space used by the application, which is obviously the physical limit defined by the microprocessor capacities.

TinyOS compiler computes the application footprint, both RAM and ROM occupation, using the specific utilities provided for each microcontroller: `avr-size` for Atmel (e.g. Mica family motes), and `msp430-size` for MSP430 (e.g. Telos motes). The result is the list of sizes statically allocated in each segment (`.data, .text, .bss`) for an object file set as parameter. An example is shown in Listing 8.1, where the command `avr-size build/mica2/main.exe` was executed.

```
text    data     bss     dec     hex filename
1500       2      47    1549     60d build/mica2/main.exe
```

Listing 8.1: List of segment sizes returned by `avr-size` utility

The static RAM memory occupied by the object file *build/mica2/main.exe* is the addition of the `.data` and `.bss` segments (49 bytes), while the ROM occupation is the addition of the `.text` and `.data` segment (1502 bytes), which coincides with the data provided by the compiler about the application footprint. Contiki compiler computes the footprint in an analogous way.

In addition to the above, the executable code is also considered. The program image size represents the amount of memory needed to download a program into the microcontroller memory during mote installation and consequently, it is limited to the available memory.

8.2.2. Portability metrics

The main metrics for determining the SN-OSAL contribution to applications portability are: the percentage of HW platforms for which code can automatically be generated, and the percentage of potential applications for which SN-OSAL is able to perform a correct translation for all OSes considered.

Table 8.1 shows a list of the platforms evaluated for the three OSes considered. When compared to Table 3.1, which represents the total number of platforms ported in each OS, it can be deduced that the percentage of supported platforms by SN-OSAL is approximately 28.5% in Contiki and 35.2% in TinyOS.

OS/Platform	TinyOS 1.x	TinyOS 2.x	Contiki
Mica	✓		
Mica2	✓	✓	
Mica2dot	✓	✓	
MicaZ	✓	✓	
Telos(A)	✓	✓	
TelosB	✓	✓	
ESB			✓
SKY			✓

Table 8.1: Relation of platforms considered for evaluation.

The second issue, computing the set of potential applications which could be written for SN-OSAL, and also successfully translated to the OSes, is a delicate task, which is difficult to determine. In order to obtain this measurement, the next assertion was taken into account: an SN-OSAL application can be ported if, and only if, the required functionality has been ported. Consider the functionalities identified in Chapter 4 for a WSN OS: *scheduling, sensors & actuators, communication, storing, debugging, clock & energy saving management*. As discussed, these services standardize the most common operations required by these kinds of devices. However, there are services that are not included, for instance, in the current implementation only a basic energy saving management has been supported. It does not prevent the API from being further extended in order to incorporate new functionalities.

8.2.3. Metrics for software cost estimation

Software cost estimation consists of determining, with a reasonable certainty, the resources of hardware and software, cost, time and effort involved in the software building. According

to [NRH+09], where a hardware cost model for sensor nodes is proposed, the WSN-specific software cost could be estimated using a suitable cost model such as COCOMO II.

The *Constructive Cost Model*, COCOMO II [BHM+00] is a mathematical model for software cost estimation and one of the most commonly applied models. COCOMO II performs the estimation using a set of parameters and values, which are established based on the experience of a great number of software projects. COCOMO II describes three models, each one more suitable to a specific development stage: *Application Composition*, *Early Design*, and *Post-Architecture*.

The *Post-Architecture* model is the most detailed and the applied in the development phase and consequently, it is the selected model for cost estimation of WSN applications. Effort is expressed as person-months (PM) and it is computed as:

$$PM = A \times (Size)^E \times \Pi_{i=1}^{17} EM_i \tag{8.1}$$

where:

- A is a constant with predefined value (A=2.94).
- E is the exponent that takes into account the features related to economies or diseconomies of scale. It is computed as:

$$E = B + 0.01 \times \Sigma_{j=1}^{5} W_j \tag{8.2}$$

  where B is predefined (B=0.91) and W_j is the weight given to each j scale factor.
- Size is a measurement of the software in KLOC, number of source lines of code (in miles).
- EM_i is every one of the seventeen effort multipliers, ranging from *Extra-high* to *Extra-low* and using values calibrated from the experience of projects.

Subsequently, for cost estimation determining the size of the application expressed as number of source lines of codes is required. According to COCOMO II, the software development effort will be estimated for some SN-OSAL applications, and compared to T1, T2 and Contiki applications.

8.2.4. Other metrics

As shown, energy constitutes a challenge for WSN applications developers because it will impact the network lifetime. There are different simulators (see Section 2.3.3.2) that provide information about the amount of work (in Joules) performed by applications per unit of time, allowing designers to measure power consumption. However, a more realistic analysis should be dynamic because it depends on the application execution and the environment conditions. Subsequently, focusing on the SN-OSAL evaluation, this metric has not been taken into account because it is not affected by code generation. In other words, SN-OSAL generates applications in a controlled environment, paying special attention to limiting the energy required.

8.3. Benchmark applications

Benchmark applications are defined since there are no standard applications for evaluation. In this section, the set of benchmark applications written for evaluating SN-OSAL are enumerated

in the following paragraphs. They are intended to cover the identified main functionalities of the OS. The key idea is to cover the simplest functions carried out by sensor nodes and to compose more complex applications using them:

1.- *Hello World* is the simplest benchmark application. Its purpose is debugging within a PC environment.

2.- *Single Timer* starts a single timer, waits for it, and finishes.

3.- *Periodic Timer* starts a periodic timer every second and waits for it. When it expires, it simply prints a message (for debugging).

4.- *Blink* is similar to *Periodic Timer*, but it toggles the red led when the timer expires.

5.- *Read Sensor* reads from a sensor, waits for the result, and prints data (for debugging).

6.- *SerialComm* creates a message and sends it over the serial port *(base station)*.

7.- *Send unicast* creates a message, and sends it to a specific address in the network.

8.- *Send broadcast* creates a message, and sends it to the entire network (broadcast address).

9.- *Periodic Send* creates a message, and periodically sends it to the entire network (broadcast address).

10.- *Send and receive* creates a message, sends it to the entire network, and waits for a response.

Figure 8.1 shows the evaluation environment for benchmark applications. On the left, all benchmark applications are programmed *ad-hoc* for every combination of operating system and sensor node. Note that the applications thus developed are platform-specific. For each platform, the source code may also differ. On the right, benchmark applications are programmed one single time using SN-OSAL, which translates automatically for the previously considered platforms.

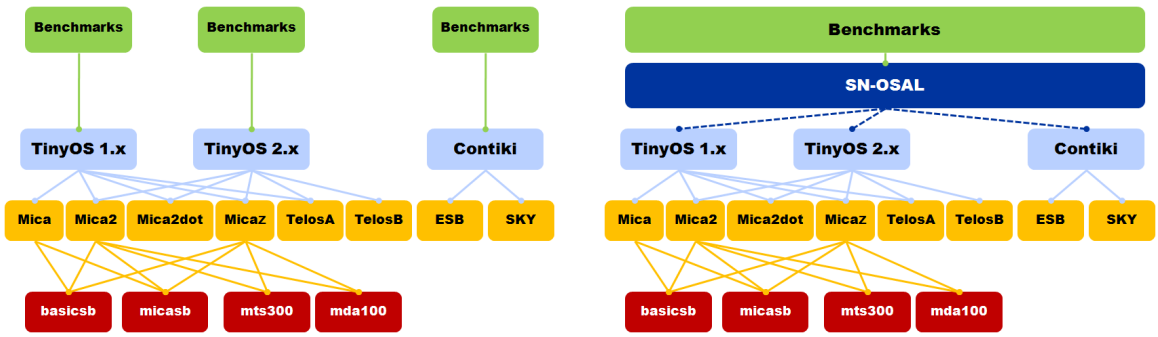

Figure 8.1: Evaluation environment. On the left, benchmarks are programmed *ad-hoc* for each particular platform. On the right, benchmarks are programmed once using SN-OSAL.

8.4. Portability evaluation: a case study

The set of benchmark applications has been developed using SN-OSAL, and subsequently, it has been generated for the three operating systems: Contiki 2.2, TinyOS 1.x and TinyOS 2.x.

The hardware platforms used to check portability are ESB, SKY for Contiki, Mica family (Mica, Mica2, Mica2dot, MicaZ), and Telos family (Telos, TelosB) for TinyOS 1.x, and Mica2, Mica2dot, MicaZ and Telos family for TinyOS 2.x. Every benchmark was ported to the complete range of platforms with no modifications, which means that SN-OSAL applications abstract away the platforms low-level details, both hardware and operating system.

As a case study, in this section one SN-OSAL application example is analyzed, *Blink*, and the equivalent applications automatically generated for Contiki and TinyOS. *Blink* is a very simple application, which toggles the red led every time that a previously configured timer expires. Listing 8.2 shows the *Blink* application using SN-OSAL.

```
1: int desc;
2: timer( int desc ) {
3:        osal_led_toggle(LED_RED);
4: }
5: main() {
6:        desc = osal_timer_start(1, SEC, REPEAT_SAMPLE);
7:        osal_task_signal(OSAL_TIMER_EVENT, desc, (void*)timer);
8: }
```

Listing 8.2: *Blink* application using SN-OSAL.

As shown, the application starts a timer every second and gets blocked until the timer fires. Setting off the timer is managed through an event handler, in which the red led is toggled. As the timer is defined as periodic (option `REPEAT_SAMPLE`) it is not necessary to start the same timer again. Therefore, the application implicitly keeps on executing an infinite loop.

Listing 8.3 depicts the application generated for Contiki operating system. As shown, the application goes into a loop where the timer is started in every iteration `(t0)`. In this way, a periodic timer is forced. The application gets blocked until the timer event is signaled, and when the timer expires, the red led is toggled. Note, there is not an event handler managing the expiration of the timer. Instead, a condition makes the protothread block until an event occurs. Then, it is evaluated if this event is due to a timer, and if so, the red led is toggled. Once it happens, the loop starts again.

```
1:  PROCESS(main, "main");
2:  AUTOSTART_PROCESSES(&main);
3:  PROCESS_THREAD(MAIN, ev, data) {
4:    static struct etimer t0;
5:    PROCESS_BEGIN();
6:    while(1){
7:         etimer_set(&t0, 1 * CLOCK_SECOND);
8:         PROCESS_WAIT_EVENT();
9:         if (ev == PROCESS_EVENT_TIMER) {
10:              leds_toggle(LEDS_RED);
11:         }
12:   }
13:   PROCESS_END();
14: }
```

Listing 8.3: *Blink* application generated for Contiki from SN-OSAL.

Listings 8.4 and 8.5 show the equivalent application for TinyOS 1.x and TinyOS 2.x respectively. Note that only the implementation file is presented and the configuration file is omitted. At

the starting point of both applications, the first thing is to start a periodic timer. The application handles the `fired` event, which is signaled by the operating system when the timer expires. If this is the case, the code associated with this timer event is executed or, in other words, the code enclosed within the event `Timer0.fired`.

```
1: implementation {
2: command result_t StdControl.init(){
3:       call Leds.init();
4:       return SUCCESS;
5: }
6: command result_t StdControl.start(){
7:       call Timer0.start(TIMER_REPEAT, 1 * 1000);
8:       return SUCCESS;
9: }
10: command result_t StdControl.stop(){
11:       return SUCCESS;
12: }
13: event result_t Timer0.fired(){
14:       Leds.redToggle();
15: }
16: }
```

Listing 8.4: *Blink* application generated for T1 from an SN-OSAL application.

Note the syntax used in both applications. In spite of identical semantics, the interfaces and names in T1 and T2 are notably different.

```
1: implementation {
2: event void Boot.booted() {
3:     call Timer0.startPeriodic (1 * 1000);
4: }
5: event void Timer0.fired(){
6:     call Leds.led0Toggle();
7: }
8: }
```

Listing 8.5: *Blink* application generated for T2 from an SN-OSAL application.

The target code is as similar as possible to SN-OSAL native code, and functionally equivalent. It means that different OS-specific applications are generated from a single, portable, and high-level SN-OSAL application. SN-OSAL offers common semantics and syntax for writing applications. The degree of equivalence between target applications is proportional to the equivalence among the OSes. Functionally, the generated applications are identical.

It is important to emphasize, the ease of development provided by SN-OSAL, which will be estimated in the next subsection. It favors a small learning curve and a reduced set of lines of source code. As can be seen, the effort of programming SN-OSAL applications is low, in comparison to the effort of learning every OS. The SN-OSAL code is very simple and development is simple and fast. The underlying operating system is abstracted away and low level implementation details are completely hidden to programmers. Applications portability is increased: from a single *Blink* SN-OSAL application, the Contiki and TinyOS specific applications for several hardware platforms are transparently and automatically obtained.

8.5. SN-OSAL overhead evaluation

This section quantifies the overhead imposed by SN-OSAL onto the target applications. SN-OSAL generates code for different WSN OSes, and it is important to know the deviation, in terms of footprint and executable size, between the generated code and the same application directly programmed on the OS. The goal is to determine the feasibility of employing SN-OSAL to program WSN applications, since high overhead will make its usage impracticable.

8.5.1. Evaluation methodology

The evaluation method determining the overhead imposed by SN-OSAL consists of the following steps:

1.- Writing the benchmark applications using SN-OSAL, and writing the benchmark applications from scratch, directly using TinyOS 1.x, TinyOS 2.x, and Contiki OSes (see Figure 8.1). Compiling the whole set of applications for the platforms cited in Table 8.1.

2.- Obtaining the three metrics for footprint analysis previously established for all test applications: RAM, ROM and executable code size.

3.- For every application and metric, comparing the measurement obtained using and not using SN-OSAL for a particular platform. A simple division of the value proceeding from the SN-OSAL and the OS-specific application will give a precise idea of the deviation introduced.

Formally, the evaluation method is summarized in Algorithm 8.1.

Algorithm 8.1 Footprint evaluation methodology

for all b in Benchmarks={ ... `The benchmarks` ... } **do**
  Write $b_{SN-OSAL}$
  for all m in Metrics={`RAM,ROM,EXE`} **do**
    $v_{SN-OSAL_{b,m}}$ = computeMetric($b_{SN-OSAL}$)
    for all o in OS={`TinyOS 1.x, TinyOS 2.x, Contiki`} **do**
      for all h in HW={`Mica,Mica2,Mica2dot,MicaZ,Telos,TelosB,ESB,SKY`} **do**
        Write $b_{o,h}$
        $v_{o_{b,h,m}}$ = computeMetric($b_{o,h}$)
        $deviation_{b,m,h,o} \leftarrow v_{SN-OSAL_{b,m}} \,/\, V_{o_{b,m,h}}$

Benchmark applications were programmed directly using TinyOS 1.x, TinyOS 2.x and Contiki for all platforms selected. As an example, consider Figure 8.2 which depicts the RAM and ROM measurements obtained for TinyOS 1.x.

Additionally, all of them were written using SN-OSAL and, automatically generated for all valid combinations previously formed between OS and HW. Figure 8.3 draws the RAM and ROM measurements associated with the target applications.

Computing the deviation for every metric is trivial once such measurements are known. The deviation gives an idea of the overhead added by the translator. To determine it, we proceeded to compare the metrics obtained from SN-OSAL benchmarks and from the benchmarks directly programmed into the OS using simple division. This comparison is denoted as *deviation* in Algorithm 8.1. Deviation values over 1.0 indicate that SN-OSAL applications exceed the corresponding

records obtained from applications directly using the underlying OS. On the contrary, values under 1.0 mean that SN-OSAL reduces the measurement obtained for the same application and platform. The following subsections show the results obtained.

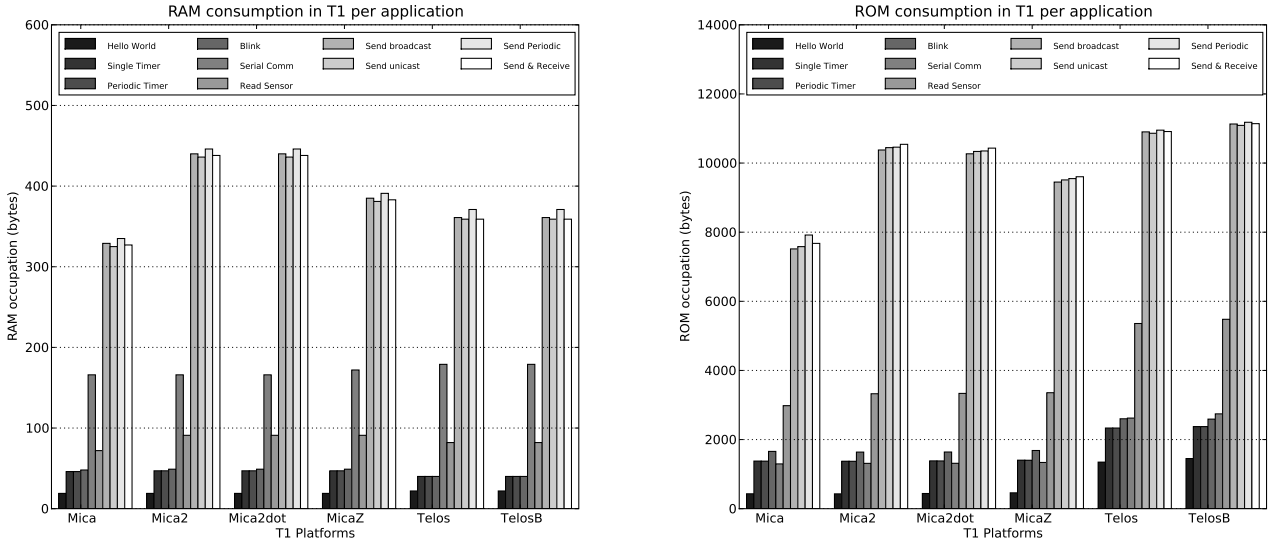

Figure 8.2: RAM and ROM consumption for benchmark applications written in T1.

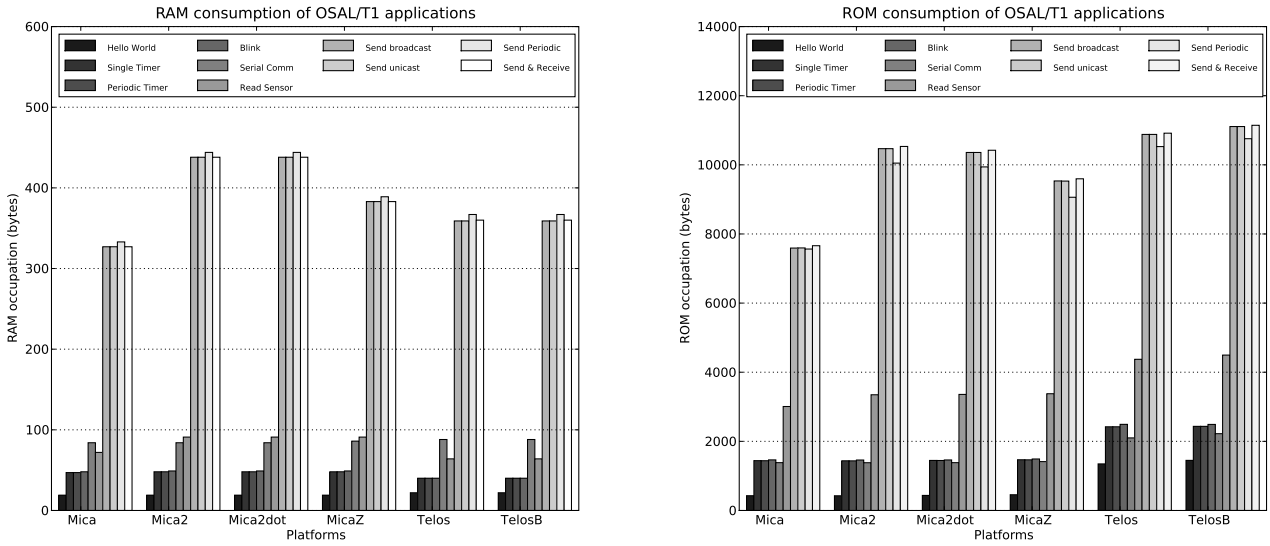

Figure 8.3: RAM and ROM consumption of benchmark applications written in SN-OSAL, and automatically generated for T1.

8.5.2. Deviation on TinyOS 1.x

Deviations were computed for every benchmark and metric over TinyOS 1.x OS. Figures 8.4 and 8.5 show the results of comparing the footprint (RAM and ROM respectively) obtained for each one written using SN-OSAL (except for *Hello World*) to the original ones written using T1, and compiled for Mica, Mica2, Mica2dot, MicaZ, Telos and TelosB sensor node platforms. The first set of graphs present the deviation values for RAM memory, and the second set, the values for ROM memory consumption. As shown, in both cases the results range between 0.81 and 1.05 for all benchmark applications, which means that a minimal redundancy of about 5% in the worst case is added in the code generation. Moreover, some applications can be reduced due to certain code optimization done during the SN-OSAL translation process, such as:

- In terms of footprint, there are TinyOS components that are functionally equivalent but more efficient than others. SN-OSAL aims to identify the most efficient component for every functionality.
- Due to the fact that code generation is controlled, there is no place for redundant code, defined variables that are non-used, or other programming errors.

The executable code size results are presented in Figure 8.6, and follow a similar trend to the footprint metric. In particular, in the worst case, the application is overloaded by 5.1%. The first conclusion extracted from these results is the high level of precision revealed by the translator between SN-OSAL and T1. There are benchmarks with no redundancy, which means that SN-OSAL produced exactly the same code as the application built in T1 from scratch.

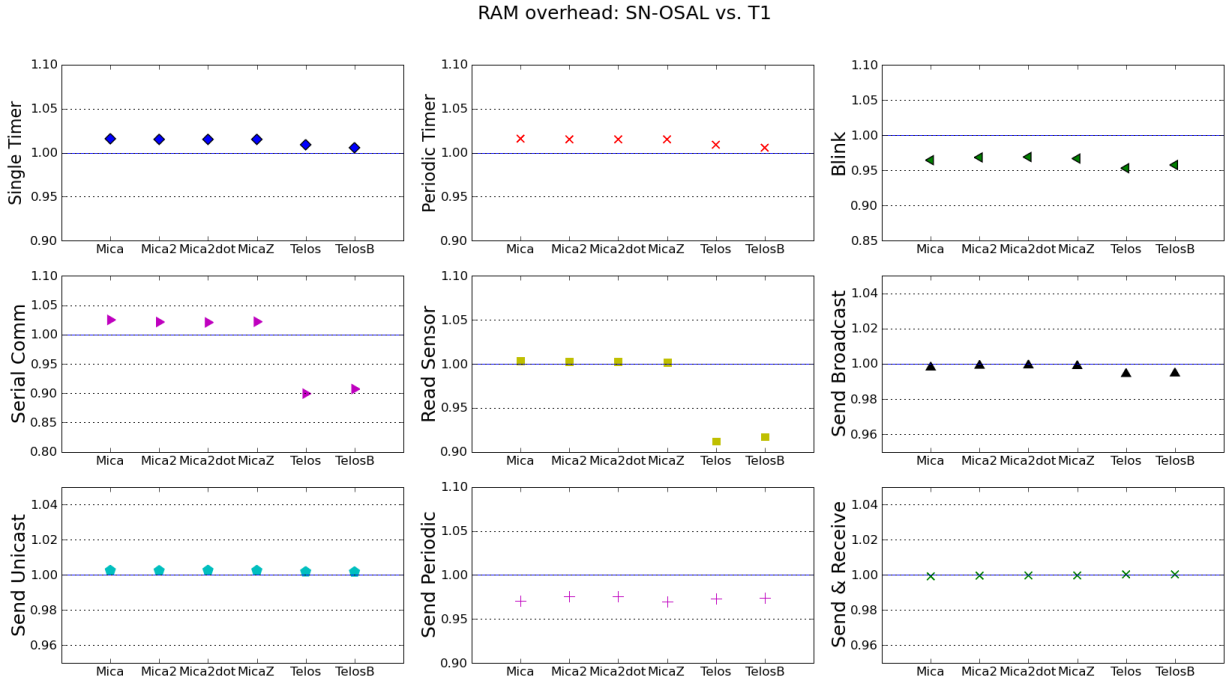

Figure 8.4: RAM overhead per platform imposed by SN-OSAL to T1.

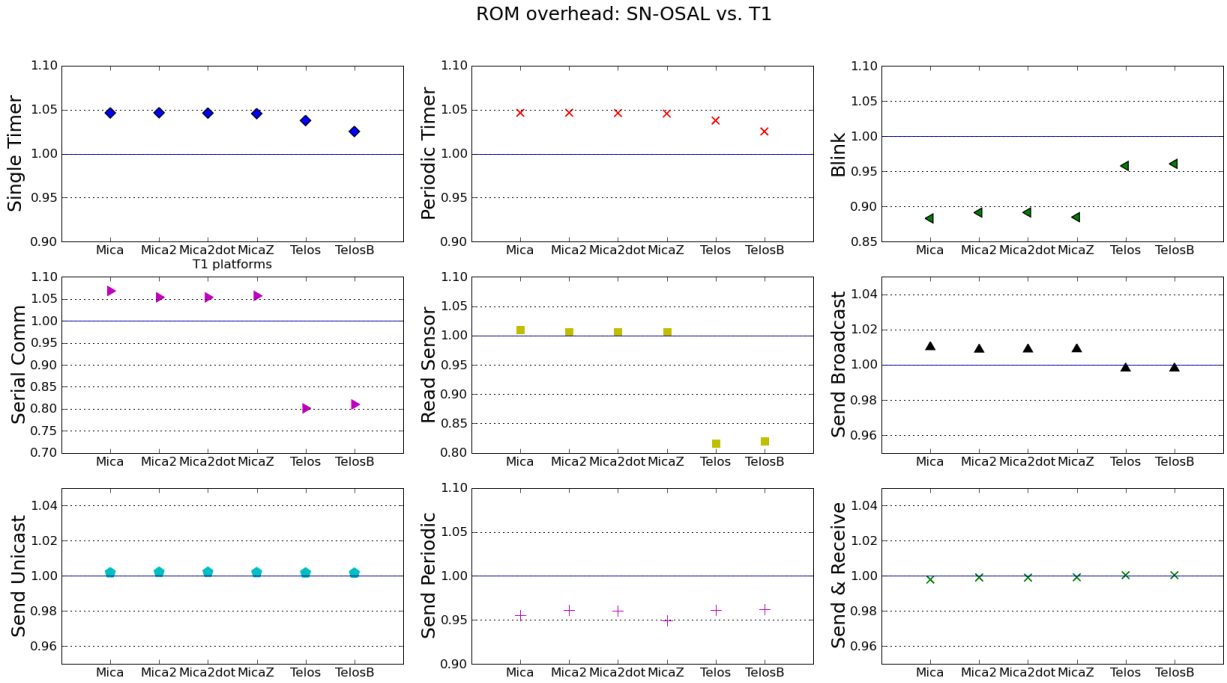

Figure 8.5: ROM overhead per platform imposed by SN-OSAL to T1.

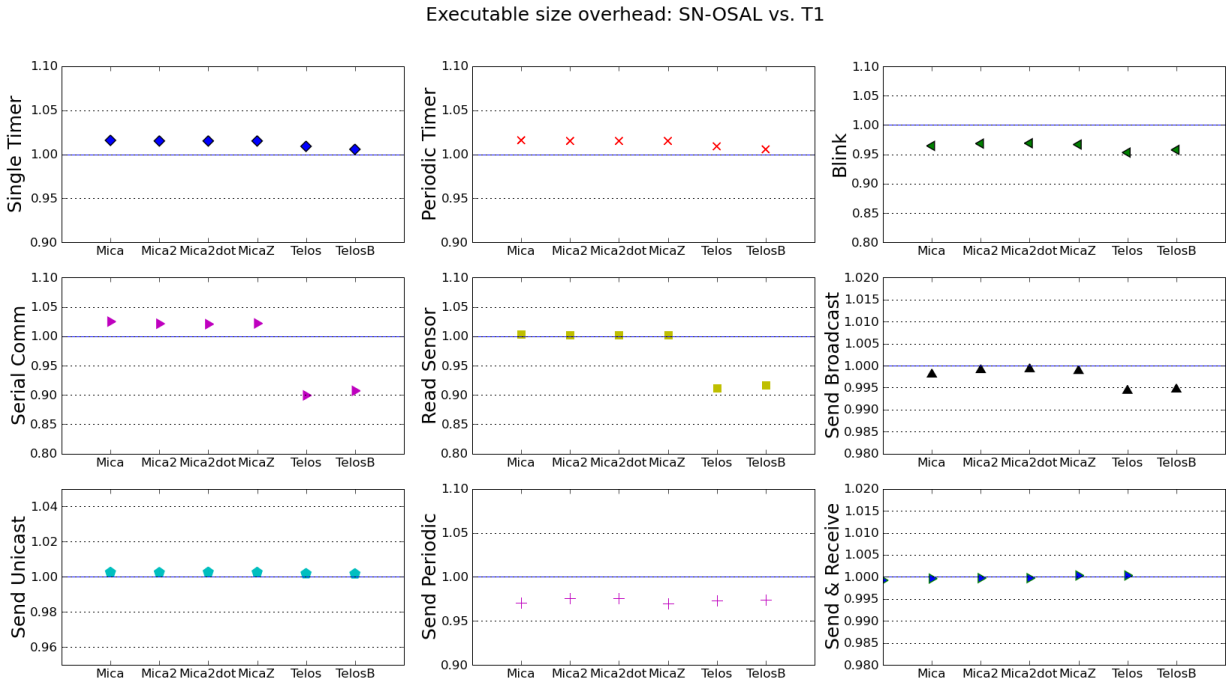

Figure 8.6: Executable size overhead per platform imposed by SN-OSAL to T1.

8.5.3. Deviation on TinyOS 2.x

Deviations were also computed for every benchmark and metric over TinyOS 2.x. Figures 8.7 and 8.8 show the results of comparing the footprint (RAM and ROM respectively) associated with SN-OSAL applications translated to T2 (except for *Hello World* application) to the original ones written using directly T2, for Mica2, Mica2dot, MicaZ, Telos and TelosB sensor node platforms. The graphics reveal that the results are better than for T1. They range between 0.63 and 1.04 for all benchmark applications, which means that, in the best case, a redundancy of about 4.6% was obtained over the original application, while in the better case it was possible to reduce by about 33.3% the RAM memory consumption for *Blink* application.

Figure 8.9 shows the comparison of the executable code size. A clearer conclusion can be reached in this case because all the considered benchmarks obtain marks less than or equal to 1.0, which means that the SN-OSAL to T2 translator is even more precise than SN-OSAL to T1 translator.

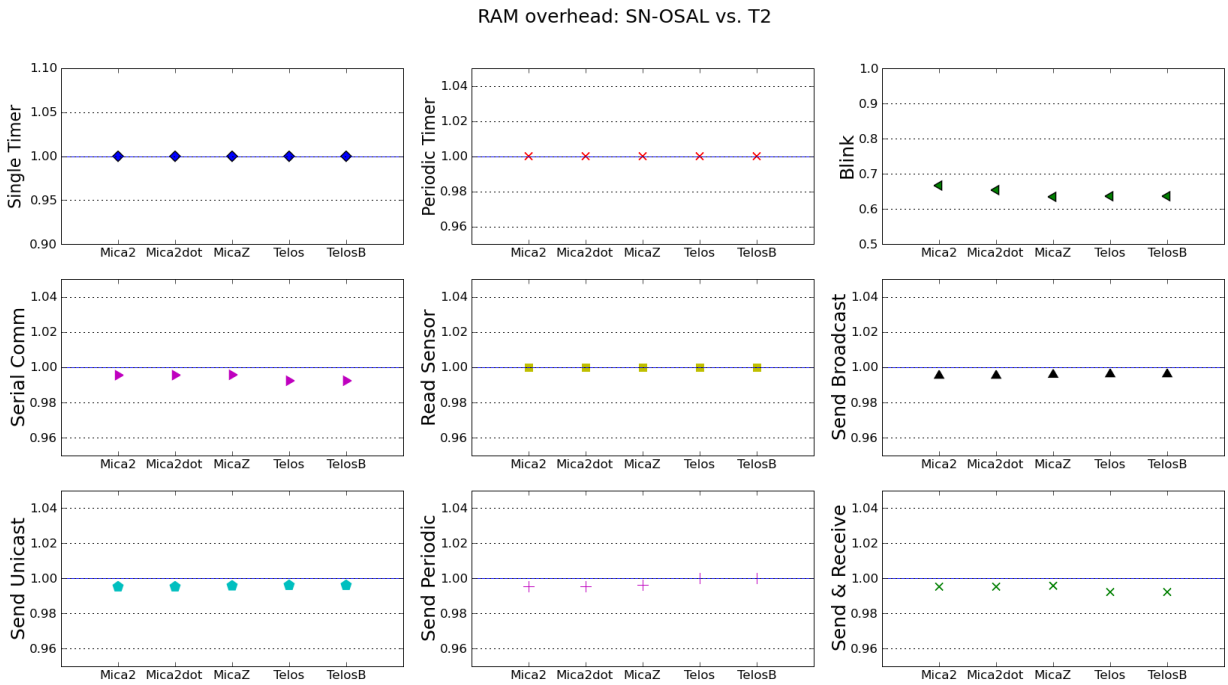

Figure 8.7: RAM overhead per platform imposed by SN-OSAL to T2.

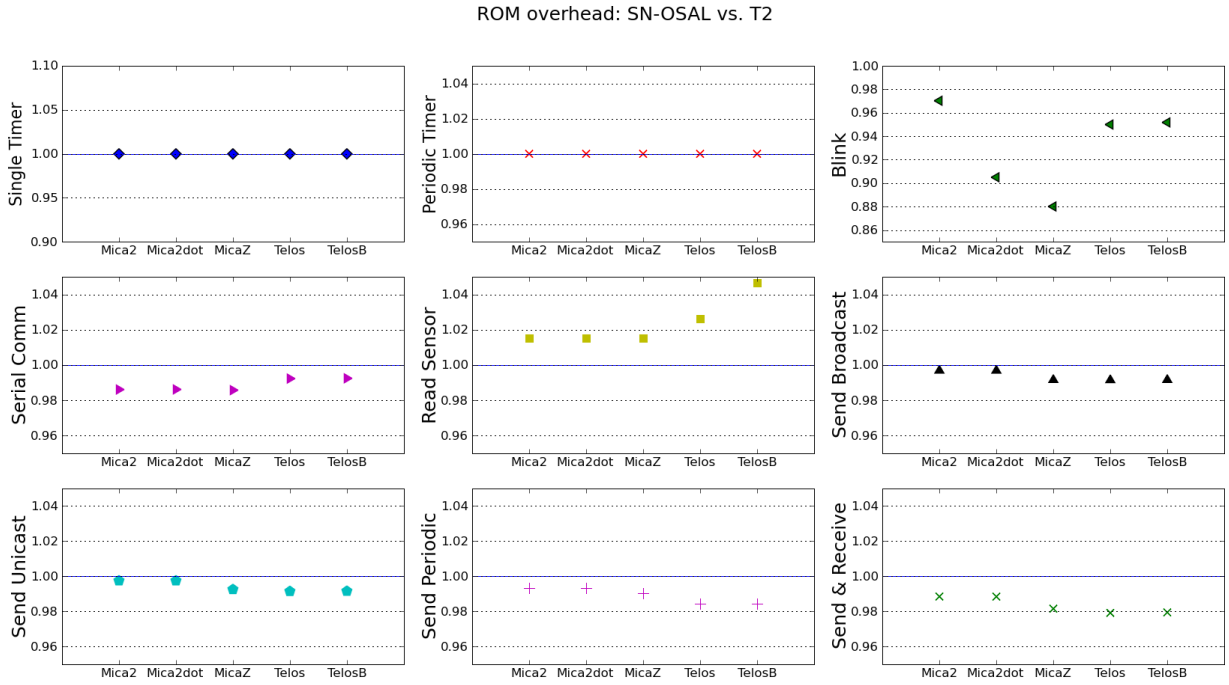

Figure 8.8: ROM overhead per platform imposed by SN-OSAL to T2.

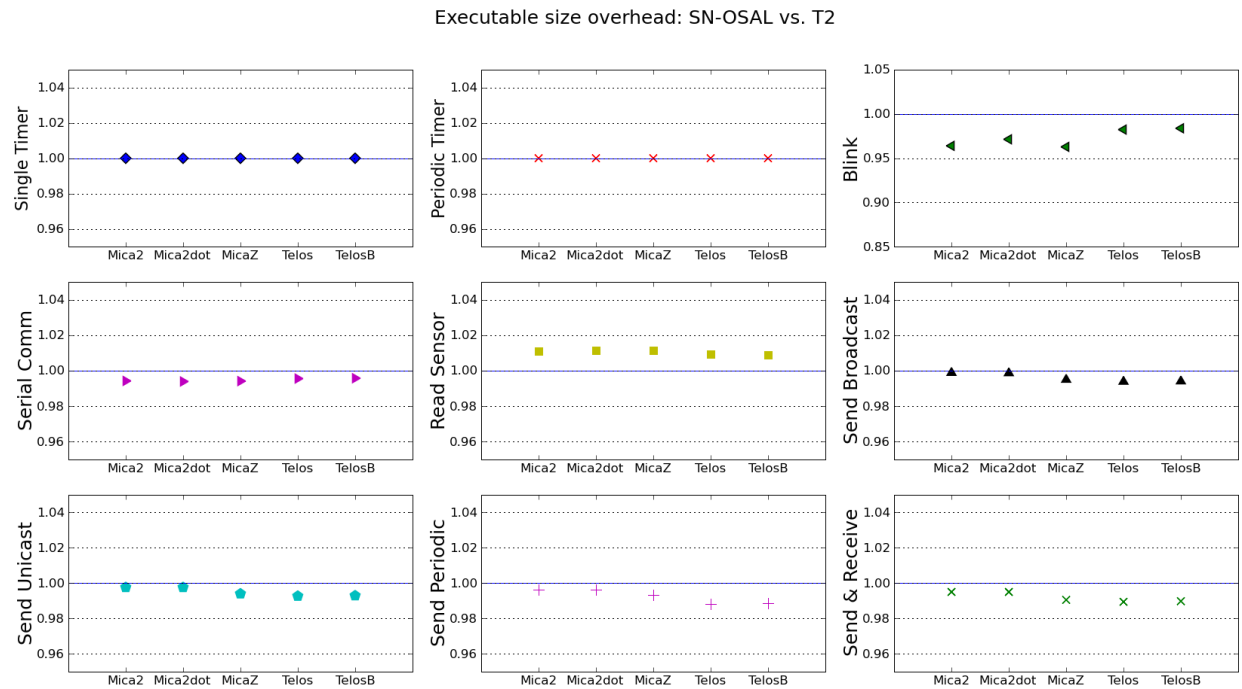

Figure 8.9: Executable size overhead per platform imposed by SN-OSAL to T2.

8.5.4. Deviation on Contiki

Finally, the deviations between Contiki OS and SN-OSAL were computed. Figures 8.10 and 8.11 show the results of comparing the RAM and ROM memory consumption due to SN-OSAL applications to the original ones written using Contiki directly, and compiled for ESB and SKY sensor node platforms. As shown, the results range between 0.9 and 1.04 for all benchmark applications. Therefore, the trend for T1 and T2 is also repeated with Contiki.

Analogously, Figure 8.12 presents the comparison of the executable size. Reduction was only achieved for a single benchmark application (*Send Unicast*), while the remaining applications fluctuate between 1.0 and 1.05, which means 5% maximal redundancy. However, in this case it is observed that the SN-OSAL to Contiki translator produces results over 0.0 for most benchmarks and consequently, the average overhead is slightly higher.

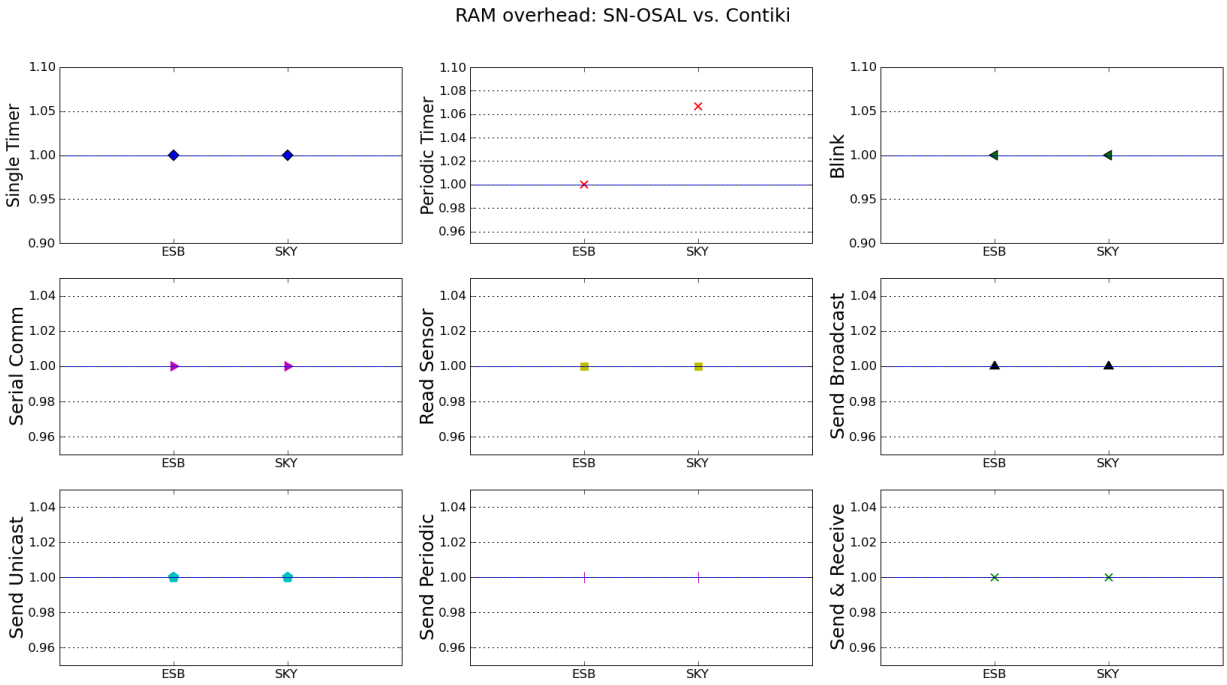

Figure 8.10: RAM overhead per platform imposed by SN-OSAL to Contiki.

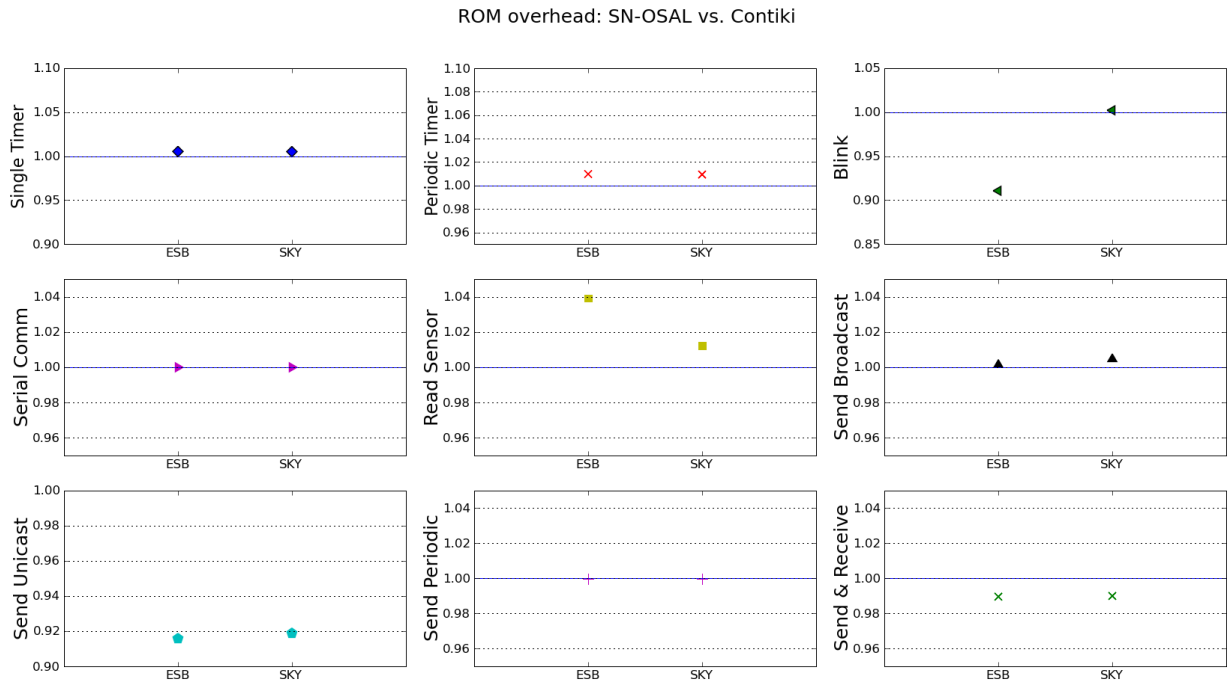

Figure 8.11: ROM overhead per platform imposed by SN-OSAL to Contiki.

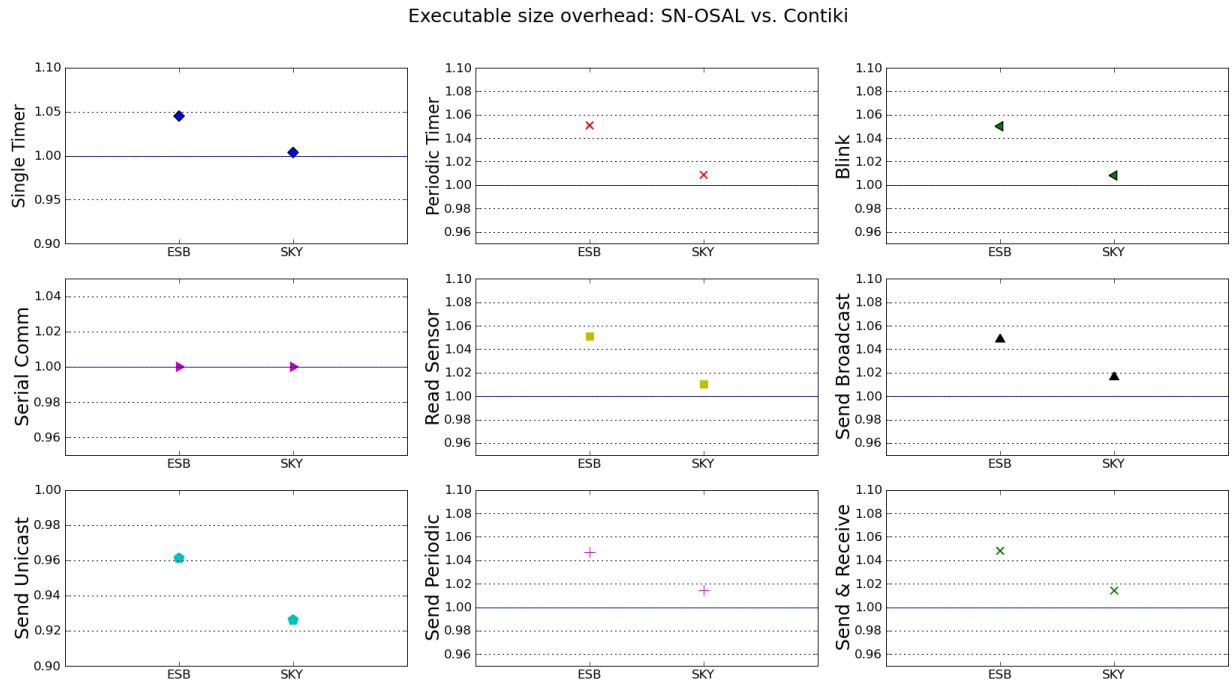

Figure 8.12: Executable size overhead per platform imposed by SN-OSAL to Contiki.

8.5.5. Feasibility conclusions

Once the overhead imposed by SN-OSAL to every valid pair {OS,HW} has been studied, this section aims to establish conclusions about the feasibility of incorporating an intermediate *Operating System Abstraction Layer* between applications and operating system in order to improve the WSN applications development process. As shown, the maximum overhead achieves the value 1.05, while the minimal overhead records 0.64 (for *Blink* benchmark, T2, all platforms). This means that the original application is reduced by almost 33%. In general, most values are quite close to 1.0. In other words, the generation code process of SN-OSAL produces applications that are very similar to the original ones with practically no redundancy. In the WSN environment, this condition of no redundancy is a key issue in determining the feasibility of using SN-OSAL.

From the previous results it is possible to extract some other conclusions. Because most measurements are kept under the limit of 1.0, a certain reduction is achieved. The average reduction percentage has been calculated in order to quantify how much RAM, ROM, and executable size is decreased (or increased) per platform on average for the three OSes. To compute the average, the 10 benchmark applications were taken into account. Note that a negative reduction means an increase.

Results appear in Figures 8.13, 8.14 and 8.15 for TinyOS 1.x, TinyOS 2.x, and Contiki operating systems, respectively. As shown in Figure 8.13, the maximum reduction achieved is for Telos family motes, and T1 OS, which registers a reduction of about 7.5%, 4% and 2.5% for RAM, ROM and executable size, respectively. RAM is the metric that has improved for all platforms, while ROM and EXE metrics remain close to the original applications.

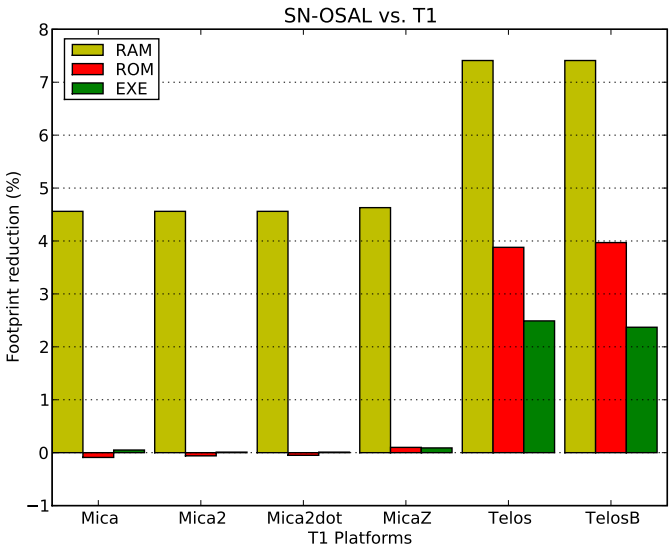

Figure 8.13: Average footprint reduction over T1.

For TinyOS 2.x the RAM metric is also improved for all target platforms, obtaining an average reduction of close to 4%. As shown in Figure 8.14, results of ROM and EXE metric also show a positive reduction for all platforms.

Figure 8.15 depicts the worst case, which is the increase of 3% for executable size metric, ESB platform and Contiki OS, which means that the SN-OSAL to Contiki translator produces, in general, an increase of the executable size for the applications. With the exception of this case, a

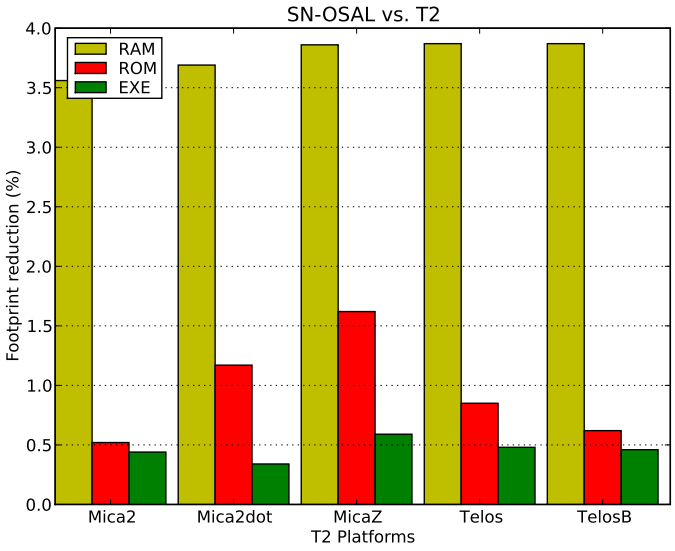

Figure 8.14: Average footprint reduction over T2.

positive average reduction is obtained for every platform.

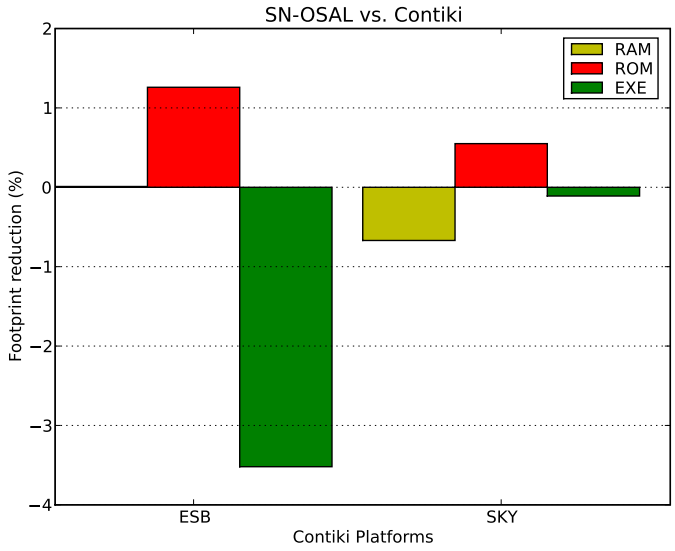

Figure 8.15: Average executable size reduction over Contiki.

8.6. Real-world applications

In addition to the benchmark applications, real-world applications must be considered in order to demonstrate the feasibility of using SN-OSAL in terms of work, portability and overhead. To achieve this, the *XMTS300* application distributed by Crossbow has been considered. This

application is available at: http://www.xbow.com/Support/wSoftwareDonwloads.aspx and it has been developed in TinyOS 1.x.

XSensorMTS300 aims to sample the sensors integrated into the MTS300 sensor board, which uses a 51-pin connector, which means that it can be coupled to Mica, Mica2 and MicaZ motes.

The behavior of *XSensorMTS300* application is simple, but several real-world applications have been built based on it, and frequently, minor changes have been required. The application samples temperature, light, battery, acceleration (x and y axis), magnetometer sensors (x and y axis) and microphone sensors sequentially every 2 seconds. Once all the measurements are collected, a message containing the sensor readings is broadcast to the network. The number of total lines of the original application is 262 (34 lines within the configuration component, and 228 lines in the implementation component), it uses 22 interfaces and involves 11 components.

8.6.1. SN-OSAL development and portability

XSensorMTS300 was developed using SN-OSAL. A unique file is required which contains 108 lines of source code, which means a 58.77% reduction from the original application. The code is clear and easy to understand because of its POSIX-style, and it fulfills the complete set of requirements of the original application.

For T1 and T2 operating systems, which have support for the mts300 sensor board, this application was automatically generated for Mica2 and MicaZ platforms, as they are compatible with that sensor board. The remainder of the motes cannot be ported: Mica2Dot does not have a 51-pin connector and the Telos family motes integrate the sensors in the board, and therefore, there is no expansion connector.

Listing 8.6 shows the fragment of the XSensorMTS300 application representing the event handlers management in SN-OSAL: sequential sensor readings.

```
1: void timerfired( int desc ) {
2:       osal_io_read(battery, data);
3: }
4: void readBattery(int sensor, int data) {
5:       osal_pkt.battery = data;
6:       osal_io_close(battery);
7:       osal_io_read(temp, data);
8: }
9: void readTemperature(int sensor, int data) {
10:       osal_pkt.temp = data;
11:       osal_io_close(temp);
12:       osal_io_read(light, data);
13:}
14: void readLight(int sensor, int data) {
15:       osal_pkt.ligth = data;
16:       osal_io_close(light);
17:       osal_io_read(light, data);
18: }
19: void readMic(int sensor, int data) {
20:       osal_pkt.mic = data;
21:       osal_io_close(accel_x);
22:       osal_io_read(accel_x, data);
23: }
24: void readAccelX(int sensor, int data) {
```

```
25:         osal_pkt.accel_x = data;
26:         osal_io_close(accel_y);
27:         osal_io_read(accel_y, data);
28}
29: void readAccelY(int sensor, int data) {
30:         osal_pkt.accel_y = data;
31:         osal_io_close(mag_x);
32:         osal_io_read(mag_x, data);
33:}
34: void readMag_x(int sensor, int data) {
35:        osal_pkt.mag_x = data;
36:        osal_io_close(mag_y);
37:        osal_io_read(mag_y, data);
38:}
39: void readMag_y(int sensor, int data) {
40:        osal_pkt.mag_y = data;
41:        osal_io_close(mag_y);
42:        osal_task_create("Send data", (void *)&send, 0);
43}
```

Listing 8.6: XSensorMTS300 fragment in SN-OSAL: Event handlers.

The two following listings present the same piece of code once it has been generated from the previous application to TinyOS 1.x and TinyOS 2.x respectively. As shown, the code obtained is functionally equivalent to the original application. In TinyOS 2.x the generated code is quite similar, except for the event prototypes whose names differ from TinyOS 1.x.

```
1: event result_t Timer0.fired(){
2:         call VoltageControl.start();
3:         call Battery.getData();
4:              return SUCCESS;
5:}
6: async event result_t Battery.dataReady(uint16_t data) {
7:         osal_pkt_new->battery = data;
8:         call VoltageControl.stop();
9:         call TempControl.start();
10:        call Temp.getData();
11:        return SUCCESS;
12:}
13: async event result_t Temp.dataReady(uint16_t data) {
14:        osal_pkt_new->temp = data;
15:        call TempControl.stop();
16:        call PhotoControl.start();
17:        call Light.getData();
18:        return SUCCESS;
19:}
20: async event result_t Light.dataReady(uint16_t data) {
21:         osal_pkt_new->light = data;
22:         call PhotoControl.stop();
23:         call MicControl.start();
24:         call Microphone.getData();
25:         return SUCCESS;
26:}
27: async event result_t Microphone.dataReady(uint16_t data) {
```

```
28:         osal_pkt_new->mic = data;
29:         call MicControl.stop();
30:         call AccelControl.start();
31:         call Accel_X.getData();
32:         return SUCCESS;
33:}
34: async event result_t Accel_X.dataReady(uint16_t data) {
35:         osal_pkt_new->accel_x = data;
36:         call AccelControl.stop();
37:         call AccelControl.start();
38:         call Accel_Y.getData();
39:         return SUCCESS;
40:}
41: async event result_t Accel_Y.dataReady(uint16_t data) {
42:         osal_pkt_new->accel_y = data;
43:         call AccelControl.stop();
44:         call MagControl.start();
45:         call Mag_X.getData();
46:         return SUCCESS;
47:}
48: async event result_t Mag_X.dataReady(uint16_t data) {
49:         osal_pkt_new->mag_x = data;
50:         call MagControl.stop();
51:         call MagControl.start();
52:         call Mag_Y.getData();
53:         return SUCCESS;
54:}
55: async event result_t Mag_X.dataReady(uint16_t data) {
56:         osal_pkt_new->mag_y = data;
57:         call MagControl.stop();
58:         post send();
59:         return SUCCESS;
60:}
```

Listing 8.7: T1 application fragment generated from SN-OSAL: Event handlers.

```
1: event void Timer0.fired(){
2:         call Battery.read();
3:              return SUCCESS;
4:}
5: event void Battery.readDone(error_t result, uint16_t val) {
6:         osal_pkt_new->battery = val;
7:         call Temp.read();
8:}
9: event void Temp.readDone(error_t result, uint16_t val) {
10:        osal_pkt_new->temp = val;
11:        call Light.getData();
12:}
13: event void Light.readDone(error_t result, uint16_t val) {
14:         osal_pkt_new->light = val;
15:         call Microphone.read();
16:}
17:event void Microphone.readDone(error_t result,uint16_t val) {
18:         osal_pkt_new->mic = val;
```

```
19:         call Accel_X.read();
20:}
21: event void Accel_X.readDone(error_t result, uint16_t val) {
22:         osal_pkt_new->accel_x = val;
23:         call Accel_Y.read();
24:}
25: event void Accel_Y.readDone(error_t result, uint16_t val) {
26:         osal_pkt_new->accel_y = val;
27:         call Mag_X.getData();
28:}
29: event void Mag_X.readDone(error_t result, uint16_t val) {
30:         osal_pkt_new->mag_x = val;
31:         call Mag_Y.getData();
32:}
33: event void Mag_Y.readDone(error_t result, uint16_t val) {
34:         osal_pkt_new->mag_y = val;
35:         post send();
36:}
```

Listing 8.8: T2 application fragment generated from SN-OSAL: Event handlers.

Due to the fact that Contiki does not have support for the mts300 sensor board, and the current version of SN-OSAL only considers the ESB and SKY platforms, the final application was not obtained for Contiki. However, the code generated by SN-OSAL would be as shown in Listing 8.9.

```
1: etimer_set(&t0, CLOCK_SECOND*2);
2:       PROCESS_WAIT_EVENT();
3: if (ev == PROCESS_EVENT_TIMER){
4:    if (data==&etimer){
5:    battery_sensor.activate();
6:    PROCESS_WAIT_EVENT();
7:    if ((ev==sensors_event) && (data==&battery_sensor)){
8:       osal_pkt_new->battery = battery_sensor.value(0);
9:       battery_sensor.deactivate();
10:      temperature_sensor.activate();
11:      PROCESS_WAIT_EVENT();
12:      if ((ev==sensors_event)&&(data==&temperature_sensor)){
13:         osal_pkt_new->temp = temperature_sensor.value(0);
14:         temperature_sensor.deactivate();
15:         light_sensor.activate();
16:         PROCESS_WAIT_EVENT();
17:         if ((ev==sensors_event) && (data==&light_sensor)){
18:            osal_pkt_new->light = light_sensor.value(0);
19:            light_sensor.deactivate();
20:            mic_sensor.activate();
21:            PROCESS_WAIT_EVENT();
22:            if ((ev==sensors_event) && (data==&mic_sensor)){
23:               osal_pkt_new->mic = mic_sensor.value(0);
24:               mic_sensor.deactivate();
25:               accel_x_sensor.activate();
26:               PROCESS_WAIT_EVENT();
27:               if ((ev==sensors_event) && (data==&
   accelx_sensor)){
```

```
28:                     osal_pkt_new->accel_x = accelx_sensor.value
    (0);
29:                     accel_x_sensor.deactivate();
30:                     accel_y_sensor.activate();
31:                     PROCESS_WAIT_EVENT();
32:                     if ((ev==sensors_event) && (data==&
    accely_sensor)){
33:                         osal_pkt_new->accel_y = accely_sensor.
    value(0);
34:                         accel_y_sensor.deactivate();
35:                         mag_x_sensor.activate();
36:                         PROCESS_WAIT_EVENT();
37:                         if ((ev==sensors_event) && (data==&
    magx_sensor)){
38:                             osal_pkt_new->mag_x = magx_sensor.
    value(0);
39:                             mag_x_sensor.deactivate();
40:                             mag_y_sensor.activate();
41:                             PROCESS_WAIT_EVENT();
42:                             if ((ev==sensors_event) && (data==&
    magy_sensor)){
43:                                 osal_pkt_new->mag_y = magy_sensor.
    value(0);
44:                                 mag_y_sensor.deactivate();
45:                                 PROCESS_YIELD();
                    //Let the protothread send executes
46:                             }
47:                         /* Close all brackets */
54:}
```

Listing 8.9: Contiki application fragment generated from SN-OSAL: Event handlers.

8.6.2. Footprint evaluation

The footprint was computed for the original and the generated applications for the valid platforms. Results are shown in Table 8.2 . As can be observed, for T1 the measurements obtained for RAM, ROM and EXE are close to the Crossbow application footprint, and the overhead ranges between 0.98 and 1.1. In the case of T2, the values obtained are higher. Note that the applications built for T2 has a bigger footprint than for T1, and, therefore, the comparison should be made to the same application developed in T2.

Metric	Crossbow		SN-OSAL/T1		SN-OSAL/T2	
	Mica2	MicaZ	Mica2	MicaZ	Mica2	MicaZ
RAM	535	535	582	567	524	593
ROM	14042	14042	13632	13812	18226	21492
EXE	24843	23281	25122	25545	38578	43401

Table 8.2: Comparison of footprint metrics among the original XSensorMTS300 application and those generated.

8.7. Software cost and productivity estimation

This section is intended to compute and analyze the cost estimation of software production using different systems: SN-OSAL, and T1, T2, and Contiki operating systems. The idea is to estimate and compare the effort required to develop one application using SN-OSAL and the effort made for programming the same one from scratch using TinyOS 1.x, TinyOS 2.x, and Contiki operating systems. As mentioned, to obtain a measurement of the software production cost, the COCOMO II model will be used.

8.7.1. Applications size

In order to use COCOMO II model, it is necessary to compute the size of the applications. The size of the application will be later used to compute the effort for each application. Table 8.3 depicts the number of source lines of code (SLOC) of every application developed using SN-OSAL, TinyOS 1.x, TinyOS 2.x and Contiki. As observed, in general the number of source lines of code required to build a SN-OSAL application is lower than in other systems. This reduction, expressed as a percentage, can be graphically viewed in Figure 8.16. As shown,

Benchmark	SN-OSAL	TinyOS 1.x	TinyOS 2.x	Contiki
Hello World	4	25	15	9
Single Timer	9	32	19	15
Periodic Timer	9	32	20	14
Blink	8	37	19	16
Serial Comm	13	50	52	12
Read Sensor	9	31	21	13
Send Broadcast	17	51	51	23
Send Unicast	14	50	52	27
Send Periodic	17	54	54	34
Send & Receive	19	54	62	30
XSensorMTS300	108	262	153	N/A

Table 8.3: Source lines of code for test applications.

Note that in TinyOS the number of lines of code is computed as the number of lines of the configuration plus the implementation component making up the application.

For TinyOS, the number of components and interfaces is also computed for every application (see Table 8.4). In addition to the application components, different components at the underlying levels must be statically wired in order to be used. Note that SN-OSAL applications are made up of a single file.

8.7.2. Effort multipliers and scale factors for WSN applications

Next, COCOMO II will be applied to compute the software cost estimation. Specifically, the *Post-Architecture* model will be selected for the computation, due to the fact that it is the one applied in the development phase after the architecture has been established. It is important to point out that the results obtained are an estimation based on the experience collected from a great number of projects, but they do not constitute an exact evaluation.

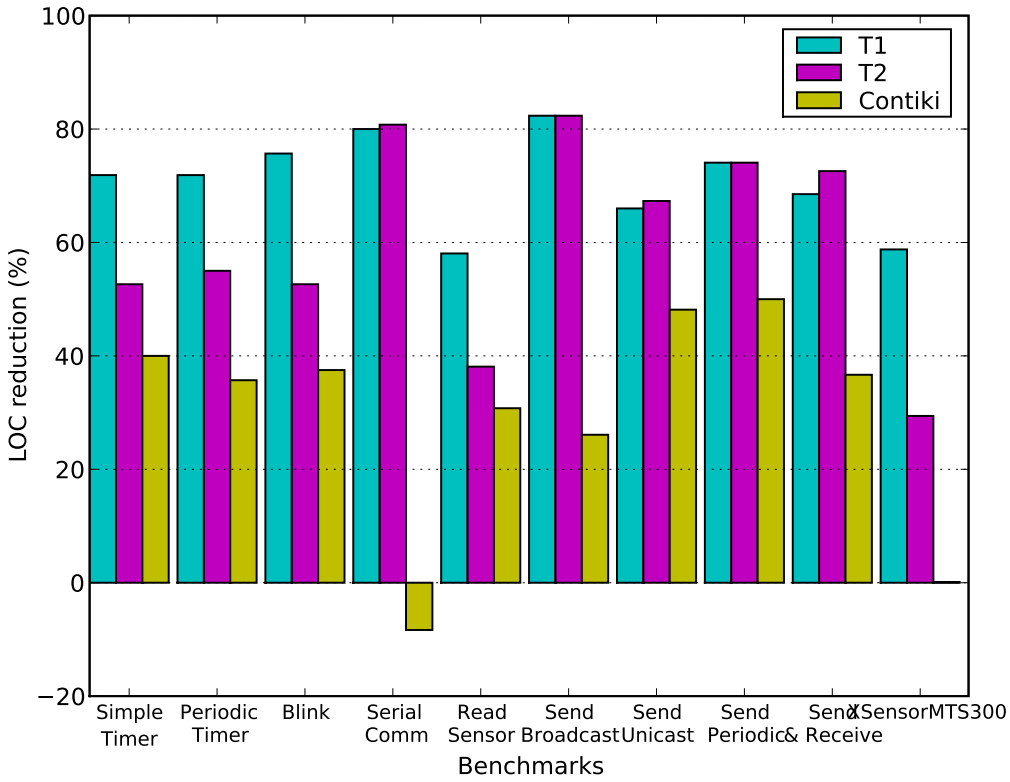

Figure 8.16: Reduction in percentage of source lines of code (SLOC).

Let $App_{SN-OSAL}$ be a generic application built using SN-OSAL, and let App_{T1}, App_{T2} and $App_{Contiki}$ be the equivalent application developed from scratch in T1, T2 and Contiki respectively. To obtain the effort of development, COCOMO II defines a set of seventeen effort multipliers that quantify different aspects of the project, such as environment, development team, or tools used. Table 8.5 shows such effort multipliers for the application developed in SN-OSAL, TinyOS 1.x, TinyOS 2.x and Contiki.

In this first approach, the effort multipliers of development using SN-OSAL are bigger than the effort in the remainder of operating systems. Reasons for this fact include the lack maturity of SN-OSAL with respect to the other environments and the subsequent inexperience of team development. Thus, SN-OSAL has been penalized in the PLEX and LTEX indicators. Given that in other environments greater maturity has been achieved, effort multipliers are lower. On the contrary, SN-OSAL allows reusability (portability) without increasing the effort performed. In this way, the PVOL and RUSE indicators have been corrected.

Subsequently, these multipliers currently make the effort of developing in SN-OSAL slightly greater than in other systems such as Table 8.5 depicts.

In summary, the following considerations were taken into account, which affect the selection of the value of the effort multipliers:

- Due to the fact that SN-OSAL means a novel way of developing applications, the experi-

Benchmark	TinyOS 1.x		TinyOS 2.x	
	Components	Interfaces	Components	Interfaces
Hello World	2	1	2	1
Single Timer	3	2	3	2
Periodic Timer	3	2	3	2
Blink	4	3	3	2
Serial Comm	3	2	3	5
Read Sensor	3	2	3	2
Send Broadcast	3	2	4	4
Send Unicast	3	2	4	4
Send Periodic	4	3	5	5
Send & Receive	3	3	5	5
XSensorMTS300	11	22	9	18

Table 8.4: Components and interfaces involved in the benchmark applications.

EM	Description	Values			
		SN-OSAL	TinyOS 1.x	TinyOS 2.x	Contiki
RELY	Required software reliability	Very High	Very High	Very High	Very High
DATA	Database size	Very Low	Very Low	Very Low	Very Low
CPLX	Product complexity	Very Low	Very Low	Very Low	Very Low
RUSE	Developed for reusability	Low	Very High	High	High
DOCU	Documentation match to life-cycle needs	Nominal	Nominal	Nominal	Nominal
TIME	Execution time constraints	Extra High	Extra High	Extra High	Extra High
STOR	Main storage constraint	Very High	Very High	Very High	Very High
PVOL	Platform volatility	Low	High	High	High
ACAP	Analyst capabilities	High	High	High	High
PCAP	Programmer capability	High	High	High	High
PCON	Personal continuity	Nominal	Nominal	Nominal	Nominal
APEX	Applications experience	High	High	High	High
PLEX	Platform experience	Very Low	Very High	High	High
LTEX	Language and tool experience	Very Low	Very High	High	High
TOOL	Use of software tools	Very Low	Very Low	Very Low	Very Low
SITE	Multisite development	Very High	Very High	Very High	Very High
SCED	Required development schedule	Very High	Very High	Very High	Very High
$\Pi Em_{i=1}^{17}$		1.371	1.097	1.183	1.183

Table 8.5: Effort multipliers of WSN applications according to COCOMO II model.

ence of the programmers is very low in relation to the experience of TinyOS and Contiki programmers (LTEX, PLEX). Only text-based programming was done, and simple text editors and compilers were used (TOOL).

- Reliability of the system is considered high. Consider, for example, medical applications (RELY).
- Given that SN-OSAL allows generating code without knowing the platform details, the effort of reusability is considered lower as in other environments (RUSE, PVOL).
- Developers implied in the project are considered expert (ACAP, PCAP).

Table 8.6 shows the weights for the five scale factors, which allow the existence of economies or diseconomies of scale to be analyzed. As shown, the value computed for the exponent E is found over 1.0, which means diseconomy of scale. This is to be expected because wireless sensor networks is a relatively new technology, where the applications building process, even that of hardware and software, is farther from being consolidated.

Scale factor	Description	Values			
		SN-OSAL	TinyOS 1.x	TinyOS 2.x	Contiki
PREC	Precedentedness	Very Low	Nominal	Nominal	Nominal
FLEX	Development Flexibility	High	High	High	High
RESL	Architecture/Risk Resolution	Very Low	Very Low	Very Low	Very Low
TEAM	Team Cohesion	Nominal	Nominal	Nominal	Nominal
PMAT	Process Maturity	Very Low	Very Low	Very Low	Very Low
$\Sigma W_{j=1}^{5}$		26.39	23.91	23.91	23.91
E		1.2739	1.2491	1.2491	1.2491

Table 8.6: Scale factors of WSN applications according to COCOMO II model.

Applying the *Post-Architecture* mathematical model, effort, the amount of calendar time in months taken to develop the product, and productivity have been estimated considering effort multipliers and scale factors computed in this section and substituting in 8.1.

8.7.3. SN-OSAL effort estimation

Knowing the size of the application in SN-OSAL, effort multipliers and scale factors, it is possible to quantify the effort expressed as person-month (PM), time of development (TDEV) and productivity (PROD). The next tables show some examples. Table 8.7 depicts these values for the benchmark application *Send & Receive*. In this example, productivity of developing in SN-OSAL is greater than in TinyOS and Contiki. The reduced number of lines of the SN-OSAL application with respect to its competitors make possible this fact.

	Effort (PM)	Time of Development (months)	Productivity (PROD)
SN-OSAL	0.038	1.240	0.494
TinyOS 1.x	0.112	1.794	0.479
TinyOS 2.x	0.142	1.937	0.434
Contiki	0.061	1.474	0.484

Table 8.7: Effort, time and productivity estimation for *Send & Receive* application

Another example is shown in Table 8.8, which shows effort, time of development and productivity for the real world *XSensorMTS300* application. In this case, effort of developing the *XSensorMTS300* application using SN-OSAL is bigger than in the other cases because the number of source lines of code.

However, note that since a single SN-OSAL application, code for T1, T2 and Contiki can be generated, simplifying the effort and subsequently increasing the productivity, because in the better case the equivalent applications are automatically obtained for three systems. Therefore, in a first approach the productivity should aggregate the individual productivities of development for T1, T2, and Contiki. Next, a more realistic approach for the productivity is computed.

	Effort (PM)	Time of Development (months)	Productivity (PROD)
SN-OSAL	0.295	2.446	0.365
TinyOS 1.x	0.692	3.258	0.378
TinyOS 2.x	0.402	2.723	0.380

Table 8.8: Effort, time and productivity estimation for *XSensorMTS300* application.

8.7.4. Phase distribution of SN-OSAL effort

As shown, effort involved in the building of a software product can be estimated as 8.1. According with COCOMO II, effort can also be distributed into the different phases of the life cycle. Thus, effort can be reformulated as:

$$Effort = \sum_{i=1}^{n} Effort_{Phase_i} \tag{8.3}$$

where i represents each phase within the set of phases defined for a specific development methodology. For example, for *Waterfall* life cycle, according with COCOMO II model, the total effort is distributed into different phases such as Table 8.9 shows.

Phase	Effort %
Plans and Requirements	7 (2-15)
Product Design	17
Programming	64-52
Integration and Test	19-31
Transition	12 (0-20)

Table 8.9: Waterfall phase distribution percentages (taken from [BHM+00]).

Note that the percentages vary depending on the size product, and *Plans and Requirements* and *Transition* phases are in addition to the 100 percent of the amount of effort estimated by COCOMO II.

In order to estimate the effort of SN-OSAL development in a more precise way, it is necessary to take into account the amount of effort that is common and specific for each target system. It is important to point out that it is assumed that the SN-OSAL application goes to be deployed in all different systems for which SN-OSAL has a translation function. Given that SN-OSAL unifies a part of the process for different platforms, there are several phases of life cycle that are common for all target platforms (they are required only once), while another are platform-specific (and therefore, activities must be repeated once per platform).

Thus, effort due to *Plans and Requirements*, *Product Design* and *Programming* phases is performed once using SN-OSAL, while *Integration and Test* phase must be repeated for each target system[1]. In this way, the total amount of effort for development a SN-OSAL application in all considered target systems can be estimated as:

$$Effort_{SN-OSAL} = \sum_{j=1}^{p} Effort_{SN-OSAL_{Phase_j}} + \sum_{i=1}^{n} Effort_{O_{i_{Tests}}} \tag{8.4}$$

[1]In a more precise approach, it even could be possible to design the tests once although the same tests have to be repeated for each system.

where *j* corresponds to the reused phases (*Plans & Requirements, Design* and *Implementation*) and *n* represents the set of target operating systems. Setting the percentages due to the different phases (see Table 8.9), the total effort is:

$$Effort_{SN-OSAL} = 0.69 \times (Effort_{SN-OSAL}) + \sum_{i=1}^{n}(0.31 \times (Effort_{O_i})) \quad (8.5)$$

From the previous formula can be deduced that the 69% (due to the SN-OSAL design and programming) is the average percentage of effort that is required to perform only once, which also means the reused effort percentage among different developments, while the remainder 31% (due to the tests) is an effort which must be repeated once per platform. It is important to point out that the percentages selected correspond to applications with less lines of code than 2 KLOC. Note that this value is an estimation, and a more optimistic approach might increase this value if part of analysis and tests phases are considered.

8.7.4.1. Productivity conclusions

Taking into account the previous consideration, this subsection computes the productivity for SN-OSAL. The formula to estimate productivity using COCOMO II can be expressed as:

$$Productivity_{SN-OSAL} = \frac{\sum_{i=1}^{n} KLOC_{SN-OSAL}}{Effort_{SN-OSAL}} \quad (8.6)$$

where $KLOC_{SN-OSAL}$ is the addition of lines generated for each O_i system and *i* can be T1, T2 and Contiki, due to SN-OSAL generates code for all these platforms. To illustrate this, Figure 8.17 shows the productivity obtained with SN-OSAL and T1, T2 and Contiki for the set of benchmark applications.

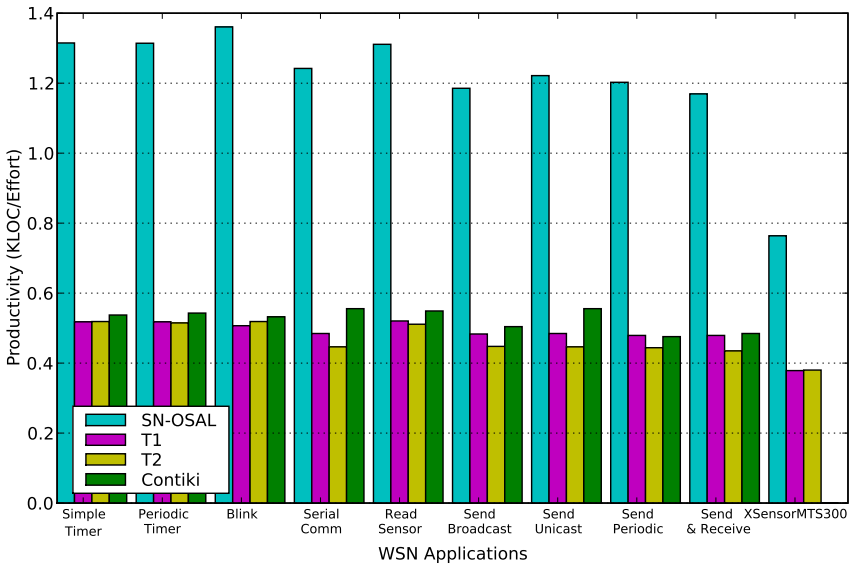

Figure 8.17: Comparison of applications productivity.

As can be observed, the productivity of SN-OSAL is bigger than other systems, because it can produce applications for different platforms, as explained in previous chapter. Subsequently, productivity can be increased using SN-OSAL. Analogously, the effort and time of development would also be decreased in a similar proportion. The productivity ratio between SN-OSAL and other environment O_j could be estimated through:

$$ratio_{SN-OSAL,O_i} = \frac{Productivity_{SN-OSAL}}{\frac{\sum_{i=1}^{n} Productivity_(O_i)}{n}} \quad (8.7)$$

where O_i corresponds to T1, T2 or Contiki operating systems. The productivity ratio reflects how much SN-OSAL improves the productivity with respect to the average productivity computed for the systems for which SN-OSAL has been ported. Figure 8.18 shows the productivity ratio per application, where the SN-OSAL productivity is compared to the average productivity for T1, T2 and Contiki.

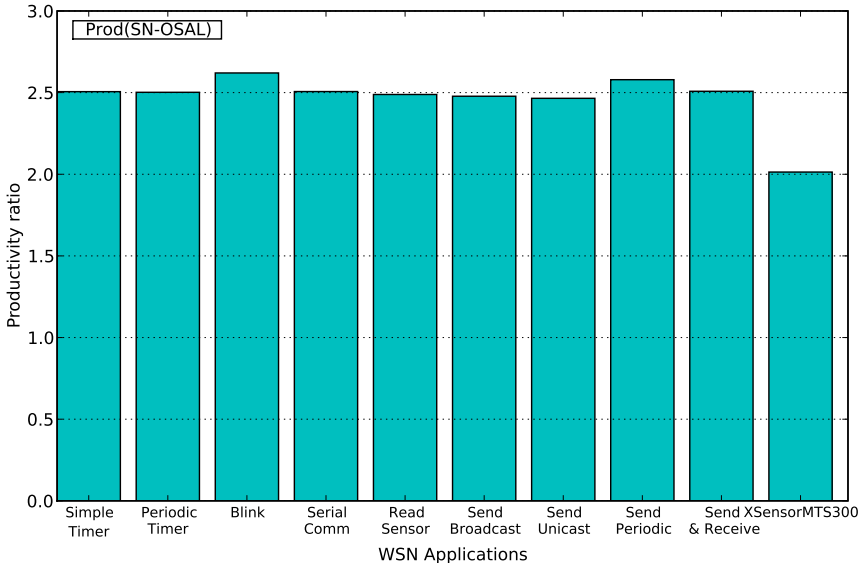

Figure 8.18: Productivity ratio between SN-OSAL and the average productivity of other systems.

8.8. SENFIS experimental results

In this section the evaluation of the SENFIS prototype is presented. SENFIS has been implemented on TinyOS and is based on the `Atmel AT45DB` [CHAa] flash memory chip, employed in Mica and TelosA motes. TinyOS provides different mechanisms of accessing the flash memory chip, depending on the abstraction level. For SENFIS, `PageEEPROM` component of TinyOS has been used, because as a low-level driver is the most efficient in terms of both energy and computation.

Table 8.10 depicts a comparison of the main settings between the file systems studied in Section 2.4 and SENFIS.

Feature	ELF	Matchbox	LiteFS	SENFIS
1	TinyOS	TinyOS	LiteOS	TinyOS
2	Mica2	Mica Family Motes	MicaZ	Mica Family Motes
3	Dynamic	Static	Dynamic	Dynamic
4	RAM, EEPROM, Flash	Flash	RAM, EEPROM, Flash	RAM, EEPROM, Flash
5	14 bytes (per flash page) 14 bytes 14 bytes per i-node (RAM)	8 bytes (per flash page)	8 bytes (per flash page) 168 bytes RAM 2080 bytes ROM	8 bytes (per flash page) 1062 bytes flash 208 bytes RAM/EEPROM
6	Unlimited	2 (Read/Write)	8	64
7	Sensor Data Configuration Data Binary program Image	Data files	Data Binary applications Device Drivers	Data stream Binary applications

Table 8.10: Comparison among different file systems for sensor nodes. Features: 1:Operating system; 2:Sensor platforms; 3:Memory allocation; 4:Memory chips used; 5:Metadata size; 6:Number of files opened; 7:Types of files.

8.8.1. Applications footprint

In order to estimate the RAM and EEPROM memory consumption of SENFIS, a simple TinyOS application containing one file open operation for SENFIS, ELF and Matchbox file systems has been compiled. Tables 8.11 and 8.12 show the memory footprint in bytes for RAM and EEPROM, respectively. As shown, SENFIS requires more RAM space than ELF and Matchbox. This is due to the fact that in the current implementation, has been simply chosen to keep a copy of the whole metadata in the RAM, therefore, the memory consumption reported here is for the worst case. However, the RAM memory space can be reduced in at least two ways. First the necessary memory can traded off with the maximal number of files. Second, the query table is not necessary in an application, which does not use TinyDB. Even though SENFIS consumes more RAM, Table 8.12 shows that the amount of employed EEPROM space is considerable lower.

Footprint (bytes)	Mica	Mica2	MicaZ	Telos
SENFIS	1282	1572	1572	1257
Matchbox	422	420	420	422
ELF	887	888	888	Platform not supported

Table 8.11: RAM footprint.

Footprint (bytes)	Mica	Mica2	MicaZ	Telos
SENFIS	1622	4812	4842	2868
Matchbox	13426	13112	13142	12358
ELF	10022	9094	9140	Platform not supported

Table 8.12: EEPROM footprint.

8.8.2. Performance evaluation

In order to evaluate the implementation of SENFIS, Avrora [Tit05] toolkit (version 1.7.106) simulator is employed. Avrora is a set of simulation and analysis tools for programs written for the AVR microcontroller produced by Atmel and the Mica2 sensor nodes. The evaluations were performed for SENFIS and ELF file systems. In the experiments SENFIS used the flash driver of TinyOS, while the ELF a dedicated flash driver implemented by the same research team.

In order to evaluate the data accesses a benchmark was developed. It generates write and read accesses of different granularities starting from 1 byte upto 16 KB (corresponding to a file of 64 flash pages). Figures 8.19 and 8.20 show the results for write and read operations of SENFIS and ELF.

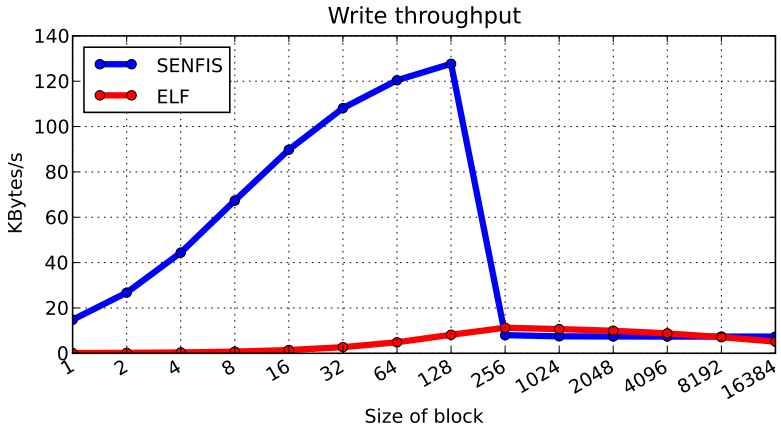

Figure 8.19: Write throughput for SENFIS and ELF for different access sizes.

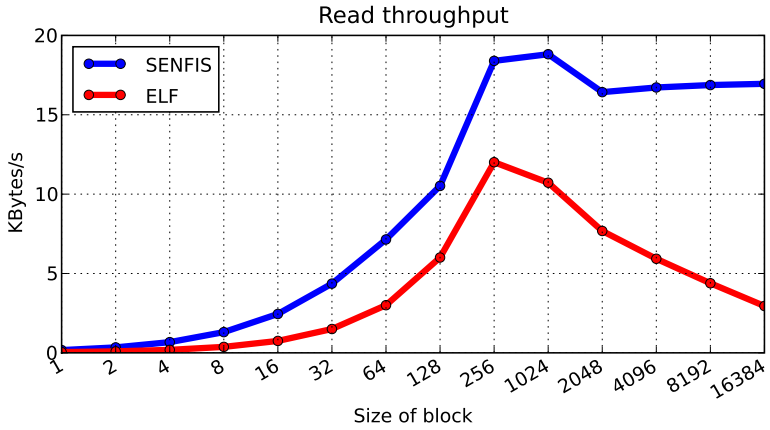

Figure 8.20: Read throughput for SENFIS and ELF for different access sizes.

In order to reduce the number of writes on flash, SENFIS aggregates small writes to the same file block in an internal buffer cache of size equal to the page size. The buffer cache is written to the flash memory either when is full or when the sensor node goes into the standby mode. This explains the better throughput of SENFIS writes with respect to ELF. For access sizes multiple of page sizes the write throughput results of SENFIS and ELF are similar.

The benefits of SENFIS simple implementation are more evident in the read throughput evaluation for both small and large access granularities. The best throughput is obtained by SENFIS for accesses, which are small multiples of flash pages, while for ELF for access sizes equal to a flash page. Table 8.13 shows the execution time in milliseconds for the other SENFIS primitives.

SENFIS primitive	Time (ms)
open	3,53
close	0,05
stat	0,134
rename	0,94
lseek	0,045

Table 8.13: Execution time of SENFIS operations.

As noticed, the more expensive operation is file open, which involves the creation of open file structures, which are immediately committed to the final storage. All the other operations have an overhead lower than 1 millisecond.

8.8.3. Energy evaluation

The direct evaluation of energy consumption with Avrora revealed a simulator bug (as of version 1.7.106). The flash energy consumption increased linearly with the simulation time, although the number of reads and writes stay constant. Additionally, the reported energy consumed by Flash appeared to be up to three orders of magnitude larger than the one defined in the specifications. Therefore, available information was used, such as the number of flash accesses and the number of CPU execution cycles in order to make a rough estimate of energy consumption.

First, the number of flash accesses that SENFIS and ELF perform for one file access was investigated. Except for small granularities, for which SENFIS accumulates all small writes to a single flash page into a buffer, SENFIS and ELF perform the same number of flash operations. Therefore, the flash energy consumption of SENFIS and ELF is roughly similar. The number of execution cycles for different access sizes, used for computing the throughput of write and read operations from Figures 8.19 and 8.20, is similar for writes and better for reads. Subsequently, ELF and SENFIS consume roughly the same amount of energy for writes, while the SENFIS spends slightly less energy for read operations, due to the smaller processor time.

8.8.4. Wear leveling evaluation

Wear leveling efficiency has been evaluation through a simulation of the SENFIS file system mounted on a Mica2 mote with 512 KB flash memory. Flash memory access pattern has been modeled for the following scenario of operations:

- Initially, the file system is empty.
- When the simulation starts, new files are created and allocated in the SENFIS file system. These files represent data produced by the sensors and stored in the local file system. The file size consists of a random integer between 1 and 32 (the maximum number of pages per segment), representing the number of 256-byte flash pages.
- When the file system utilization reaches 60%, for each new created file, an existing file is randomly selected and deleted. With this strategy, the dynamic behavior of a lossy storage model [GGP+03] has been simulated. For this model the stored data have a limited lifetime depending on their characteristics. The lifetime is modeled by means of a random function.

- For every 10000 accesses, a 24 KB file is created. This file represents the dynamic reprogramming during the mote operational life. These files are deleted immediately after their creations, given that are transferred to the mote internal memory.

- This procedure is repeated until one flash memory page is accessed 10000 times, reaching the limit of its operational life. Once this condition is reached, the simulation ends.

Figure 8.21 shows the results of our simulation in terms of number of accesses of each flash page. As observed, all of them have a similar number of accesses, very close to the limit of accesses.

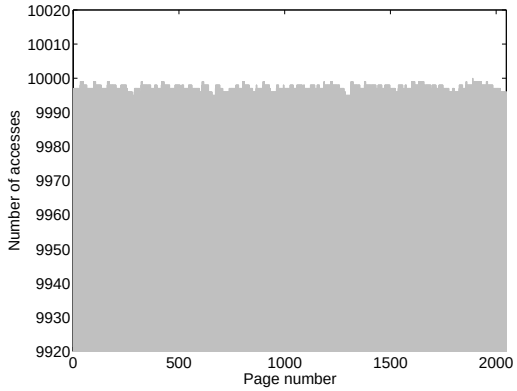

Figure 8.21: Number of accesses per page.

Figure 8.22 shows the histogram of accesses for flash pages. Note that the dispersion of the accesses is very small. More specifically, it is obtained a mean value of 9997.4 with a standard deviation of 0.8982. Based on these results, it is possible to conclude that for the considered scenario, the wear leveling policy of SENFIS obtains a uniform and distributed use of the flash memory close to optimum values.

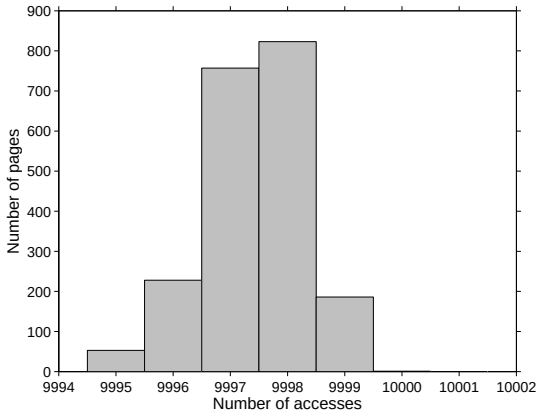

Figure 8.22: Histogram of accesses for memory pages.

8.9. Chapter summary

In this chapter the benchmarking applications and metrics have been defined in order to carry out the evaluation of SN-OSAL. The results obtained allow us to demonstrate, in terms of ease of programming, portability, and productivity, the feasibility of employing SN-OSAL to build WSN applications.

SN-OSAL applications can be described in very few lines of code. Development is simple, clear and fast, and it considerably reduces the learning curve associated with programming using every OS directly. Applications are developed without taking into account the details of the underlying platform (operating system and sensor node), and it is automatically transported to a reasonable broad set of hardware platforms. Subsequently, SN-OSAL increases the applications portability.

Furthermore, overhead has also been measured through a set of benchmark applications, which have been programmed using the three OSes directly, and also using SN-OSAL, to later be automatically generated for all possible pairs of type OS-sensor node platform considered. The results demonstrate that SN-OSAL is efficient in terms of redundancy over the original applications for most of the test scenarios. Moreover, it has been possible to obtain a certain reduction of the applications footprint and executable code size, due to code optimizations such as the selection of the most efficient components or variables management. In accordance with the results obtained, it has been possible to determine the SN-OSAL feasibility and efficiency.

Using COCOMO II, it has been demonstrated that SN-OSAL increases productivity in the development process for wireless sensor networks. At the beginning, SN-OSAL has certain disadvantages with respect to the operating systems (lack of maturity and inexperience of the development team). However, as shown, the code generation allows productivity to be computed as the addition of the individual productivities of the platforms for which the application generation was possible and correct.

Additionally, SENFIS experimental results have been presented in this chapter. Evaluation shows that SENFIS wear leveling technique provides uniform distribution of flash accesses. Read and write throughput have been quantified and compared to ELF file system. SENFIS implementation makes efficient use of resources in terms of energy consumption, memory footprint, and flash wear leveling while achieving execution times similar to existing WSN file systems.

Chapter 9

Conclusions

Keeping in mind the objectives established at the beginning of this document, preliminary results allow us to affirm that these objectives have been achieved: *A generic sensor node-centric architecture for portable applications in heterogeneous wireless sensor networks has been proposed, thus increasing the productivity, portability, and simplifying the development process in heterogeneous environments. The architecture proposed allows generic applications to be written in a POSIX-based style, and automatically translated to different WSN operating systems. This translation process increases productivity, which will be reflected in the usability of the technology.*

This chapter contains the summary and major conclusions obtained from this thesis work. Firstly, the theoretical contributions are reviewed. Secondly, practical contributions are presented, such as publications or merits related to this work. Finally, some discussion on possible future work is analyzed in order to identify some interesting research directions.

9.1. Theoretical contributions

This doctoral thesis presents a theoretical contribution to the state of the art of software development in wireless sensor networks by proposing the formalization of a sensor node-centric architecture. This architecture alleviates the effort of learning and developing associated with the heterogeneity and complexity exhibited by sensor nodes and WSN operating systems. These contributions can be summarized as follows:

- The description of the architecture has fulfilled the MDA standard principles. MDA encourages to define independent models of the systems, plus a *transformation process* which allows to obtain platform specific models. This description is a novel proposal that allows the concerns about components specification and instantiation to be separated, also clearly formulating the integration method for them. Thus, it makes possible to redefine or customize the traditional architecture, always respecting the established principles.

- The mathematical description of the sensor node-centric architecture. Theory of sets has been used to formalize the different layers of the architecture, the relation between them, and the *transformation process* between generic and specific applications. This formalization also constitutes a novelty, and it allows the requirements of the architecture to express unambiguously.

Focusing on the instantiation concerns, the development of a complete architecture according to the formalization posed has been carried out in this thesis. Two upper layers to be located on top of the traditional architecture were designed, implemented and integrated together: an *Operating System Abstraction Layer* and an *Application Layer*. They mask the heterogeneity imposed by sensor nodes, and provide programmers with an easy, fast, and unambiguous method for applications development in both design and implementation stages, without renouncing control over applications.

Specifically, the previous ideas have been developed in this thesis through the next contributions:

- *Sensor node-centric architecture design.*
  The design of the sensor node-centric architecture has been accomplished in a multi-layered approach, where each layer interacts only with the immediate upper and lower one through a well-defined interface. It has allowed us to study each abstraction level in a specific way, and to focus our attention on the interfaces for connecting them. We have assumed the existence of traditional architecture composed of hardware and operating system, on which two additional levels have been incorporated: *Operating System Abstraction Layer* and *Application Layer* with well distinguished responsibilities and interfaces. These layers are intended to hide the complexity of the underlying platform for programmers. Additionally, physical devices have been described through XML manifests and schemas, which allow the contents specification to be decoupled from the instantiation. Three operating systems have been extensively studied in terms of functionality and interface: TinyOS 1.x, TinyOS 2.x and Contiki.

- *Architecture formalization using mathematical notation.*
  The description of the complete architecture using the theory of sets has contributed to sit the basis for building a generic and flexible architecture, where each layer is viewed as a reusable component that must fulfill the restrictions formulated in such description. In particular, two functions were clearly stated: a *portability* and a *translation* function between HW and OS, and OS and OSAL respectively. The translation function forces the definition of an OSAL component that export an interface, which encapsulates and standardizes the services offered by the set of WSN OSes considered. The process of translation has been mathematically formalized through a composite function able to obtain the specific application from a high-level application.

- *Design and implementation of SN-OSAL.*
  Sensor Node Open Services Abstraction Layer (SN-OSAL) has been designed and implemented as an instance of an *Operating System Abstraction Layer* component on top of the traditional architecture. Its goal is to offer a set of POSIX-based services that can be demultiplexed into OS-specific calls. The mapping rules ($\nearrow_{\mathcal{T}}$) among SN-OSAL and the three previously mentioned OSes were presented in Appendix B of this document. SN-OSAL also provides a pre-compiler (denominated `osalc`) that prevents badly-formed applications. The component carrying out the *translation process* was denominated *Translation Engine*. This performs the automatic generation of the code equivalent from a unique, generic, portable, and POSIX-based application written on top of SN-OSAL for a range of hardware platforms and three different OSes.

- *Domain Specific Language to describe applications on top of the architecture.*
  At the highest abstraction level in the architecture proposed, the *Application Layer*, a DSL

to describe generic applications on top of SN-OSAL has been designed. This DSL is denominated *Sensor Node Domain Specific Language* (SN-DSL). It has been elaborated to uncouple the programming language used by the underlying OS from the applications programming, and to hide thus their execution models. Name space, syntax, and writing rules for SN-DSL have been detailed. With the aim of formalizing its syntax, a context-free grammar (or type-2 in Chomsky hierarchy) able to generate it has been presented using BNF notation. SN-DSL is also POSIX-based and subsequently, applications writing results clear and simple.

- *Graphical framework for WSN applications development using SN-OSAL.*
  Once a textual notation for applications programming was stated through SN-DSL, a graphical framework for developing SN-OSAL applications using a visual notation was presented in Chapter 7. This development framework was denominated *VisualOSAL*. It constitutes a complete IDE and provides a semi-automatic support for the WSN applications life cycle, using the architecture proposed. Several outputs are generated: high-level code (encoded in SN-DSL), operating system-specific source files and the final executable code ready to be downloaded into the target sensor node.

- *Evaluation of SN-OSAL through a set of metrics and applications.*
  SN-OSAL was evaluated in terms of portability, overhead and productivity. For the three evaluations different metrics were consciously described: applications and platforms for which SN-OSAL can automatically generate code; footprint and executable code size; and effort, time of development and productivity.

  Experiments for overhead measuring consisted of comparing the results obtained for RAM, ROM and executable size metrics, for applications built from scratch using the native OS directly, and the applications automatically generated by SN-OSAL for these OSes. The preliminary results concluded that the overhead imposed by SN-OSAL is minimal, even it was possible to reduce such metrics for some platforms due to code optimizations.

  Estimation of productivity was carried out applying the COCOMO II model. Using this model, the results that were obtained shown that the initial effort for building SN-OSAL applications is greater than the effort of development in TinyOS or Contiki, due to the immaturity of the system. However, due to the fact that SN-OSAL automatically generates code for different platforms the productivity can be computed as the aggregate of the individual productivities. Therefore, the hypothesis established at the beginning of this document has been successfully validated.

- *SENsor Node File System (SENFIS) for wireless sensor networks applications.*
  To show the flexibility of incorporating new components or improving the existing ones, a file system for WSNs was developed: SENsor Node File System (SENFIS). SENFIS was intended to be a prototype showing the feasibility of extending the architecture proposed. It was conceived to substitute the Matchbox file system released in the first version of TinyOS. SENFIS addresses both scalability and reliability concerns. Design and implementation details of SENFIS were described. SENFIS can be mainly used in two broad scenarios. First, it can be employed transparently as a permanent storage for distributed TinyDB queries, in order to increase reliability and scalability. Second, it can be directly used by a WSN application for permanent storage of data on the WSN nodes. The experiments show that SENFIS implementation makes efficient use of resources in terms of energy consumption, memory footprint, flash wear leveling, while achieving execution times similar to existing WSN file systems.

9.2. Practical contributions

During the development of this thesis work several practical contributions related to this research field were made. These practical contributions are classified into three groups: publications, technology transfer, and other merits.

9.2.1. Publications

Publications present the main theoretical ideas and results shown in this thesis. They are classified into three groups: journals, and international conferences and national conferences. Following, the derived publications are listed:

- Journal publication:
  - SENFIS: a SENsor Node File System for increasing the scalability and reliability of wireless sensor networks applications.
    ISSN: 0920-8542 (Print) 1573-0484 (Online). DOI 10.1007/s11227-009-0275-8.
    Journal: The Journal of SuperComputing.
    Date: April, 2009.

- International conference publications:
  - A lightweight storage file system for sensor nodes.
    ISBN: 1-60132-082-5, 1-60132-083-3 (1-60132-084-1).
    Conference: Second International Workshop on Scalable Data Management, Applications and Systems [SDMAS 2008] within Parallel and Distributed Processing Techniques and Applications [PDPTA 2008].
    Date: July, 2008.
  - A MDA-based development framework for sensor networks applications.
    ISSN: 0302-9743.
    Conference: 4th IEEE International Conference on Distributed Computing on Sensor Systems [DCOSS 2008].
    Date: June, 2008.
  - A driver model based on Linux for TinyOS.
    ISBN: 1-4244-0840-7. IEEE Catalog Number: 07EX1633C.
    Conference: IEEE Second International Symposium on Industrial Embedded Systems [SIES'2007].
    Date: July, 2007.
  - Deconstructing the Wireless Sensor Networks Architecture.
    ISBN: 1-4244-0777-X. IEEE Catalog Number: 06EX1451.
    Conference: IEEE First International Symposium of Embedded Systems [IES' 2006].
    Date: October, 2006.
  - Data Driven Infrastructure and Policy Selection to Enhance Scientific Applications in Grid.
    ISSN: 0302-9743.
    Conference: Scientific Applications of Grid Computing: First International Workshop, SAG 2004.
    Date: 2004.

- National conference publication:
  - Acabando con los desarrollos *ad-hoc* en Wireless Sensor Networks.
    ISBN: 84-690-0551-0.
    Conference: XVII Jornadas de Paralelismo [JP 2006].
    Date: September, 2006

9.2.2. Technology transfer

A research and development project related to monitoring agricultural process, with application to vineyards, was completely implemented using WSN technology. The details of the project are shown below:

- Control Automatizado de Procesos Agrícolas (COPA).
  Code: 2008/00244/001.

Other research projects were developed:

- Sistema escalable de gestión de entrada/salida.
  Code: 2008/00055/001
- Nuevas técnicas de almacenamiento escalable en computación de altas prestaciones.
  Code: 2007/04320/001
- Técnicas de optimización y fiabilidad para sistemas de entrada/salida escalables de altas prestaciones.
  Code: 2006/03515/001.

9.2.3. Other merits

- Research stay in the "Sensor Networks and Pervasive Computing Group" of the University of Bonn (Germany) under the supervision of Dr. Pedro José Marrón, over the period of March 31^{st} to July 30^{th}, 2008.
- Workshop chair and organizer committee member of *International Workshop on Wireless Sensor Networks Architectures, Simulation and Programming* (WASP'09) within the MOBILWARE international conference, held in Berlin, 2009.
- Several Degree Projects focused on different aspects of WSNs have been directed, receiving all special distinction:
  - *Monitorización de la temperatura de un edificio mediante una red de sensores inalámbrica usando motes MicaZ*, developed by Rubén García Olalla (University Carlos III de Madrid).
  - *Design, implementation and evaluation of a Wireless Sensor Network using GSM/GPRS technology*, developed by Gabriel Heredia Palacios (University Carlos III de Madrid).
  - *A graphical tool for development of TinyOS-based Wireless Sensor Networks applications*, developed by Giacomo Tartari (University of Bologna).
  - *Design and implementation of a flash file system based on TinyOS for Mica family motes*, developed by Stefano Lama (University of Bologna).

9.3. Future work

The work described in this thesis could be further extended and improved in many different aspects. This section presents the future work derived from this thesis.

- Regarding sensor node-centric architecture:
  - Analysis, design, and implementation of network services adding new capabilities and features to sensor nodes which are programmed using SN-OSAL. In particular, a Network Time Protocol (NTP) and a code dissemination protocol for remote programming are currently being developed. In this way, the sensor node-centric architecture proposed scales to network levels (macroprogramming) by offering control and management services, which could be easily linked to SN-OSAL programs. Figure 9.1 depicts the system overview. Applications writing is accomplished on top of the architecture proposed using SN-OSAL (as described in this thesis), and subsequently, a unique application version is available, which could be shared and distributed among different developer teams.

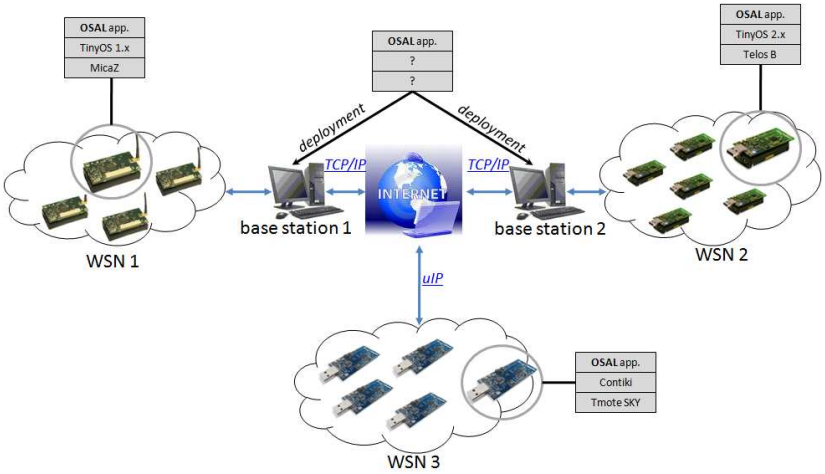

Figure 9.1: Network services monitoring by SN-OSAL. SN-OSAL application will include both the application itself as well as different network services which can be monitored from the base station (or even gateway).

The transformation process between the high-level application and the target platform is performed by SN-OSAL as explained in this document. In this way, SN-OSAL acts as an integrator element among different network technologies. Low-level details about network protocols and physical radios are kept hidden under SN-OSAL. During the services transcription, SN-OSAL must take into account the physical details of source nodes in order to incorporate the required components and primitives for them. In the current prototype, network services are limited to a NTP and a dissemination code protocol, but it does not prevent other network services such as debugging from being added. The PC communicating to the sensor network also assumes the role of network coordinator, which carries out control and monitoring functions. As in the node-centric programming, the macroprogramming should be performed in a transparent way to programmers.

  - Deployment and evaluation of the architecture in large heterogeneous networks.

- Regarding *Application Layer*:
  - In spite of the fact that a grammar has been created to specify the SN-DSL syntax, the behavior of applications could also be defined using a Formal Description Technique (FDT) such as *Language of Temporal Ordering Specification* (LOTOS) [ED89]. In this way, the functionality of SN-OSAL applications could be clearly stated.
  - *VisualOSAL* has been presented as a development framework for graphical composition using SN-OSAL. However, although it is in an advanced state of development, some features shown in this document have not been completely finished.
- Regarding *Operating System Abstraction Layer*:
  - The set of services provided by the *Sensor Node Open Services Abstraction Layer* can be further extended. This is necessary to meet new requirements of sensor nodes. This is similar to the way in which the first version of POSIX was released several times to support new extensions. Specifically, more mechanisms for saving energy should be incorporated. These services would take advantage of microprocessor skills, and the knowledge of the application itself to intelligently update the energy level in which the microprocessor works.
  - Increasing the portability degree by extending the range of platforms for which SN-OSAL generates code. It implies incorporating new sensor nodes and operating systems, and, subsequently, expanding the translation function ($\diagdown\!\!\diagdown_{\mathcal{T}}$) and the translation process carried out by SN-OSAL. Good candidates for this purpose would be Mantis and the motes that it supports.

Appendix A

Hardware description

A.1. DTD example: A thermistor sensor

```
<?xml version="1.0" encoding="UTF-8"?>
<!--DTD generated by ... -->
<!ELEMENT Manifest (definition, signal, assemblies, interface, resource*, constraints*)>
<!ATTLIST Manifest
        Signature CDATA #REQUIRED
>
<!ELEMENT SENSOR (Manifest)>
<!ELEMENT action (parameter*)>
<!ATTLIST action
        name CDATA #REQUIRED
        fileAction CDATA #REQUIRED
        ReturnValueType CDATA #REQUIRED
>

<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
        index CDATA #REQUIRED
        type CDATA #REQUIRED
>
<!ELEMENT assemblies (file)>
<!ELEMENT board EMPTY>
<!ATTLIST board
        model CDATA #REQUIRED
>
<!ELEMENT constraints (#PCDATA)>
<!ELEMENT definition (model, vendor, type, sensorboard, reading)>
<!ATTLIST definition
        datasheet CDATA #REQUIRED
>
<!ELEMENT export (operation)>
<!ELEMENT file EMPTY>
<!ATTLIST file
        nameAssemby CDATA #REQUIRED
        provider CDATA #REQUIRED
        fileAssembly CDATA #REQUIRED
>
<!ELEMENT interface (export)>
<!ELEMENT location EMPTY>
<!ATTLIST location
        name CDATA #REQUIRED
>
<!ELEMENT model (#PCDATA)>
<!ELEMENT operation (action+)>
<!ELEMENT output EMPTY>
<!ATTLIST output
        name CDATA #REQUIRED
>
<!ELEMENT power EMPTY>
<!ATTLIST power
        name CDATA #REQUIRED
>
<!ELEMENT reading (units, scale, resistance, value)>
<!ELEMENT resistance EMPTY>
<!ATTLIST resistance
        ohms CDATA #REQUIRED
        constant CDATA #REQUIRED
        variable CDATA #REQUIRED
>
<!ELEMENT scale EMPTY>
<!ATTLIST scale
        size CDATA #REQUIRED
```

```
>
<!ELEMENT value EMPTY>
<!ATTLIST value
        min CDATA #REQUIRED
        max CDATA #REQUIRED
>
<!ELEMENT sensorboard (board+)>
<!ELEMENT signal (output, location, power)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT units EMPTY>
<!ATTLIST units
        name CDATA #REQUIRED
        out CDATA #REQUIRED
>
<!ELEMENT vendor (#PCDATA)>
```

Listing A.1: Characterizing one thermistor sensor by DTD.

A.2. XML Schema example: The MTS300 sensor board

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSpy v2006 sp2 U (http://www.altova.com)-->
<!DOCTYPE sensorboard SYSTEM "SensorBoard.dtd">
<sensorboard>
        <Manifest signature="MTS300CA-0000000001">
        <definition>
        <model>MTS300CA</model>
        <vendor>Crossbow Technology</vendor>
        <platform>
                <mote name="mica" file="http://www.xbow.com/mica_datasheet.html"/>
                <mote name="mica2" file="http://www.xbow.com/mica2_datasheet.html"/>
                <mote name="micaZ" file="http://www.xbow.com/micaZ_datasheet.html"/>
        </platform>
        <connectionSchema fileConnection="connectionSchema-MTS300CA.txt">
                <pin number="1" signal="GND ANALOG" function=""/>
                <pin number="2" signal="VDD ANALOG" function=""/>
                <pin number="3" signal="INT 3" function=""/>
                <pin number="4" signal="INT 2" function=""/>
                <pin number="5" signal="INT 1" function=""/>
                <pin number="6" signal="INT 0" function=""/>
                <pin number="7" signal="DC BOOT SHUTDOWN" function=""/>
                <pin number="8" signal="LED 3" function=""/>
                <pin number="9" signal="LED 2" function=""/>
                <pin number="10" signal="LED 1" function=""/>
                <pin number="11" signal="RD" function=""/>
                <pin number="12" signal="WR" function=""/>
                <pin number="13" signal="ALE" function=""/>
                <pin number="14" signal="PW7" function=""/>
                <pin number="15" signal="FLASH CLK" function=""/>
                <pin number="16" signal="PROG MOSI SPI" function=""/>
                <pin number="17" signal="PROG MISO SPI" function=""/>
                <pin number="18" signal="SCK SPI" function=""/>
                <pin number="19" signal="FLASH SO" function=""/>
                <pin number="20" signal="FLASH SI" function=""/>
                <pin number="21" signal="I2C BUS 1CLK" function=""/>
                <pin number="22" signal="I2C BUS 1DATA" function=""/>
                <pin number="23" signal="PWM0" function=""/>
                <pin number="24" signal="PWMIA" function=""/>
                <pin number="25" signal="AC+" function=""/>
                <pin number="26" signal="AC-" function=""/>
                <pin number="27" signal="UART RXD0" function=""/>
                <pin number="28" signal="UART TXD0" function=""/>
                <pin number="29" signal="PW0" function=""/>
                <pin number="30" signal="PW1" function=""/>
                <pin number="31" signal="PW2" function=""/>
                <pin number="32" signal="PW3" function=""/>
                <pin number="33" signal="PW4" function=""/>
                <pin number="34" signal="PW5" function=""/>
                <pin number="35" signal="PW6" function=""/>
                <pin number="36" signal="ADC7" function=""/>
                <pin number="37" signal="ADC6" function=""/>
                <pin number="38" signal="ADC5" function=""/>
                <pin number="39" signal="ADC4" function=""/>
                <pin number="40" signal="ADC3" function=""/>
                <pin number="41" signal="ADC2" function=""/>
                <pin number="42" signal="ADC1" function=""/>
                <pin number="43" signal="ADC0 BBOUT" function=""/>
                <pin number="44" signal="LITTLE GUY RESET" function=""/>
                <pin number="45" signal="LITTLE GUY SPI CLOCK" function=""/>
                <pin number="46" signal="LITTLE GUY MISO" function=""/>
                <pin number="47" signal="LITTLE GUY MOSI" function=""/>
                <pin number="48" signal="RESET" function=""/>
                <pin number="49" signal="PWMAB" function=""/>
                <pin number="50" signal="VCC" function=""/>
                <pin number="51" signal="GND" function=""/>
                <pin number="52" signal="NOT USED" function=""/>
                <pin number="53" signal="NOT USED" function=""/>
```

```
        </connectionSchema>
        <sensor>
                <item type="Microphone" signatureSensor="0000A-0000000001"/>
                <item type="Sounder" signatureSensor="0000A-0000000002"/>
                <item type="Light" signatureSensor="0000A-0000000003"/>
                <item type="Thermistor" signatureSensor="0000A-0000000004"/>
                <item type="Accelerometer" signatureSensor="0000A-0000000005"/>
                <item type="Magnometer" signatureSensor="0000A-0000000006"/>
        </sensor>
        </definition>
        <assemblies>
                <file nameAssemby="Test Firmware" provider="Crossbow Technology" fileAssembly="MTS300CA/
                    Assemblies/firmware_test.c"/>
                <file nameAssemby="TinyOSDriver" provider="UC Berkeley" fileAssembly="MTS300CA/driver_tinyos.c"
                    />
                <file nameAssemby="MOSDriver" provider="" fileAssembly="MTS300CA/driver_mos.c"/>
        </assemblies>
        <interface>
        <import>
                <componentImport nameComponent="Sensor" signatureComponent="0000A-0000000001"/>
                <componentImport nameComponent="Sensor" signatureComponent="0000A-0000000002"/>
                <componentImport nameComponent="Sensor" signatureComponent="0000A-0000000003"/>
                <componentImport nameComponent="Sensor" signatureComponent="0000A-0000000004"/>
                <componentImport nameComponent="Sensor" signatureComponent="0000A-0000000005"/>
                <componentImport nameComponent="Sensor" signatureComponent="0000A-0000000006"/>
                <componentImport nameComponent="ADC" signatureComponent="0000B-0000000001"/>
        </import>
        <export>
                <operation>
                <action name="setOnSounder" fileAction="MTS300CA/Operations/setOnSounder.c" ReturnValueType=""/
                    >
                <action name="setOnMicrophone" fileAction="MTS300CA/Operations/setOnMicrophone.c"
                    ReturnValueType=""/>
                <action name="setOnAccelerometer" fileAction="MTS300CA/Operations/setOnMicrophone.c"
                    ReturnValueType=""/>
                <action name="setOnMagnometer" fileAction="MTS300CA/Operations/setOnMagnetometer.c"
                    ReturnValueType=""/>
                <action name="setOnTemperature" fileAction="MTS300CA/Operations/setOnTemperature.c"
                    ReturnValueType=""/>
                <action name="setOnPhotocell" fileAction="MTS300CA/Operations/setOnPhotocell.c" ReturnValueType
                    =""/>
                <action name="setOffSounder" fileAction="MTS300CA/Operations/setOffSounder.c" ReturnValueType="
                    "/>
                <action name="setOfficrophone" fileAction="MTS300CA/Operations/setOffMicrophone.c"
                    ReturnValueType=""/>
                <action name="setOffAccelerometer" fileAction="MTS300CA/Operations/setOffMicrophone.c"
                    ReturnValueType=""/>
                <action name="setOffMagnometer" fileAction="MTS300CA/Operations/setOffMagnetometer.c"
                    ReturnValueType=""/>
                <action name="setOffTemperature" fileAction="MTS300CA/Operations/setOffTemperature.c"
                    ReturnValueType=""/>
                <action name="setOffPhotocell" fileAction="MTS300CA/Operations/setOffPhotocell.c"
                    ReturnValueType=""/>
                <action name="setOnBoard" fileAction="MTS300CA/Operations/setOnBoard.c" ReturnValueType=""/>
                <action name="setOffBoard" fileAction="MTS300CA/Operations/setOffBoard.c" ReturnValueType=""/>
                <action name="reset" fileAction="MTS300CA/Operations/reset.c" ReturnValueType=""/>
                <action name="readADC0" fileAction="MTS300CA/Operations/readADC0.c" ReturnValueType=""/>
                <action name="readADC1" fileAction="MTS300CA/Operations/readADC1.c" ReturnValueType=""/>
                <action name="readADC2" fileAction="MTS300CA/Operations/readADC2.c" ReturnValueType=""/>
                <action name="readADC3" fileAction="MTS300CA/Operations/readADC3.c" ReturnValueType=""/>
                <action name="readADC4" fileAction="MTS300CA/Operations/readADC4.c" ReturnValueType=""/>
                <action name="readADC5" fileAction="MTS300CA/Operations/readADC5.c" ReturnValueType=""/>
                <action name="readADC6" fileAction="MTS300CA/Operations/readADC6.c" ReturnValueType=""/>
                <action name="readADC7" fileAction="MTS300CA/Operations/readADC7.c" ReturnValueType=""/>
                <action name="controlI2CCLK" fileAction="MTS300CA/Operations/controlI2CCLK.c" ReturnValueType="
                    "/>
                <action name="controlI2CDATA" fileAction="MTS300CA/Operations/controlI2CDATA.c" ReturnValueType
                    =""/>
        </operation>
        </export>
        </interface>
        <resources>
                <hardwareResource index="0" name="memory" signature="0000C-0000000001"/>
                <hardwareResource index="1" name="microcontroller" signature="0000D-0000000001"/>
        </resources>
        <compatibility>
                <component signature="0000A-0000000001"/>
                <component signature="0000A-0000000002"/>
                <component signature="0000A-0000000003"/>
                <component signature="0000A-0000000006"/>
                <component signature="MICA-0000000001"/>
                <component signature="MICA2-000000002"/>
           <component signature="MICAZ-0000000003"/>
                <?Sensor?><note contract="INT1 OR INT2"></note>
        </compatibility>
        </Manifest>
</sensorboard>
```

Listing A.2: Characterizing one MTS300CA sensor board by XML Manifest.

Appendix B

Mapping functions

B.1. TinyOS 1.x prerequisites

B.1.1. List of components, wirings, and interfaces in T1

Service (S)	TinyOS 1.x interface (I)	TinyOS 1.x component (C)	TinyOS 1.x wiring (W)[1]	Platform
*	Leds	LedsC	X.Leds → LedsC	
start	Timer	TimerC	Main.StdControl → TimerC	
*	Timer as TimerID	TimerC	X.TimerID → TimerC.Timer[unique("Timer")]	
*	PowerManagement	HPLPowerManagementM	X.PowerManagement → HPLPowerManagementM	
getData	ADC as LightTSR	HamamatsuC	X.LightTSR → Hamamatsu.TSR	Telos
getData	ADC as LightTSR	HamamatsuC	X.LightPAR → Hamamatsu.PAR	Telos
getData	ADC as Humidity	HumidityC	X.Humidity → HumidityC.Temperature	Telos
getData	ADC as Temp	PhotoTemp	X.Temp → PhotoTemp.ExternalTempADC	MTS300
getData	ADC as Light	PhotoTemp	X.Light → PhotoTemp.ExternalPhotoADC	MTS300
getData	ADC as Accel_X	Accel	X.Accel_X → Accel.AccelX	MTS300
getData	ADC as Accel_Y	Accel	X.Accel_Y → Accel.AccelY	MTS300
getData	ADC as Mag_X	Mag	X.Mag_X → Mag.MagX	MTS300
getData	ADC as Mag_Y	Mag	X.Mag_Y → Mag.MagY	MTS300
getData	ADC as Microphone	MicC	X.Microphone → MicC	MTS300
*	Mic	MicC	X.Mic → MicC	MTS300
getData	ADC as Temp	SensirionHumidity	X.Temp → SensirionHumidity.Temperature	MICAWBDOT
getData	ADC as Humidity	SensirionHumidity	X.Humidity → SensirionHumidity.Humidity	MICAWBDOT
getData	ADC as LightTSR	TaosPhoto	X.LightTSR → TaosPhoto.ADC[0]	MICAWBDOT
getData	ADC as LightPAR	TaosPhoto	X.LightPAR → TaosPhoto.ADC[1]	MICAWBDOT
getData	ADC as Pressure	IntersemaPressure	X.Pressure → IntersemaPressure.Pressure	MICAWBDOT
getData	ADC as Temp	Temp	X.Temp → Temp	BASICSB
getData	ADC as Light	Photo	X.Light → Photo	BASICSB
getData	ADC as Battery	VoltageC	X.Battery → VoltageC	
*	ADCError	HumidityC	X.ADCError → HumidityC.TemperatureError	Telos
*	ADCError	HumidityC	X.ADCError → HumidityC.HumidityError	Telos
*	ADCError	SensirionHumidity	X.ADCError → SensirionHumidity.TemperatureError	MICAWBDOT
*	ADCError	SensirionHumidity	X.ADCError → SensirionHumidity.HumidityError	MICAWBDOT
*	ADCControl	ADCC	X.ADCControl → ADCC	

Continued on Next Page...

Table B.1 – Continued

Service (S)	TinyOS 1.x interface (I)	TinyOS 1.x component (C)	TinyOS 1.x wiring (W)	Platform
open	FileRead	Matchbox	X.FileRead → Matchbox.FileRead[unique (”FileRead”)]	
open	FileWrite	Matchbox	X.FileWrite → Matchbox.FileWrite[unique (”FileWrite”)]	
close	FileWrite	Matchbox	X.FileWrite → Matchbox.FileWrite[unique (”FileWrite”)]	
read	FileRead	Matchbox	X.FileRead → Matchbox.FileRead[unique (”FileRead”)]	
append	FileWrite	Matchbox	X.FileWrite → Matchbox.FileWrite[unique (”FileWrite”)]	
sync	FileWrite	Matchbox	X.FileWrite → Matchbox.FileWrite[unique (”FileWrite”)]	
readNext	FileDir	Matchbox	X.FileDir → Matchbox.FileDir	
rename	FileRename	Matchbox	X.FileRename → Matchbox.FileRename	
delete	FileDelete	Matchbox	X.FileDelete → Matchbox.FileDelete	
send	Send as SendID	MultiHopRouter	X.SendID → MultiHopRouter.Send[unique(”SendMsg”)]	
send	SendMsg as SendID	GenericComm	X.SendID → GenericComm.SendMsg[unique(”SendMsg”)]	
send	SendMsg as SendID	GenericCommPromiscuous	X.SendID → GenericCommPromiscuous. SendMsg[unique(”SendMsg”)]	
send	BareSendMsg	UARTNoCRCPacket	X.BareSendMsg → UARTNoCRCPacket	
receive	ReceiveMsg	GenericCommPromiscuous	X.ReceiveMsg → GenericCommPromiscuous. ReceiveMsg[unique(”ReceiveMsg”)]	
receive	ReceiveMsg	GenericComm	X.ReceiveMsg → GenericComm.ReceiveMsg[unique(”ReceiveMsg”)]	
receive	ReceiveMsg	UARTNoCRCPacket	X.ReceiveMsg → UARTNoCRCPacket	

Table B.1: List of interfaces (I), components (C) and wirings (W) required by a T1 program using service (S).

[1]X represents the name of the application implementation component.

B.1.2. List of events in T1

Interface (I)	Events (E)
Timer	event void fired()
ADC	async event result_t dataReady(uint16_t data)
ADCError	event result_t error(uint8_t token)
FileRead	event result_t opened (filesize_t filesize, fileresult_t result)
FileRead	event result_t readDone (void *buffer, filesize_t n, fileresult_t result)
FileWrite	event result_t opened (filesize_t filesize, fileresult_t result)
FileWrite	event result_t closed (fileresult_t result)
FileWrite	event result_t appened (void *buffer, filesize_t nWritten, fileresult_t result)
FileWrite	event result_t synced (fileresult_t result)
FileDir	event result_t nextFile (const char* filename, fileresult_t result)
FileRename	event result_t renamed(fileresult_t result)
FileDelete	event result_t deleted(fileresult_t result)
Send	event result_t sendDone(TOS_MsgPtr msg, result_t result)
SendMsg	event result_t sendDone(TOS_MsgPtr msg, result_t result)
BareSendMsg	event result_t sendDone(TOS_MsgPtr msg, result_t result)
ReceiveMsg	event TOS_MsgPtr receive(TOS_MsgPtr data)

Table B.2: List of events (E) to be implemented when a T1 application uses the interface (I)

B.2. TinyOS 2.x prerequisites

B.2.1. List of components, wirings, and interfaces in T2

Service (S)	TinyOS 2.x interface (I)	TinyOS 2.x component (C)	TinyOS 2.x wiring (W)	Platforms
*	Leds	LedsC	X.Leds → LedsC	
*	Timer<Milli> as TimerID[2]	new TimerMilliC()	X.TimerID → TimerMilliC	
*	McuSleep	McuSleepC	X.McuSleep → McuSleepC.McuSleep	
*	McuPowerState	McuSleepC	X.McuPowerState → McuSleepC.McuPowerState	
*	Read<uint16_t> as Temp	new SensirionSth11C()	Temp = SensirionSth11C.Temperature	Telos
*	Read<uint16_t> as Humidity	new SensirionSth11C()	Humidity = SensirionSth11C.Humidity	Telos
*	Read<uint16_t> as LightPar	new HamamatsuS1087ParC()	LightPAR = HamamatusuS1087ParC.Read	Telos
*	Read<uint16_t> as LightTsr	new HamamatusuS10871TsrC()	LightTSR = HamamatusuS10871TsrC.Read	Telos
*	Read<uint16_t> as Temp	new SensorMts300C() as mts300	X.Temp → mts300.Temp	MTS300
*	Read<uint16_t> as Light	new SensorMts300C() as mts300	X.Light → mts300.Light	MTS300
*	Read<uint16_t> as Sounder	new SensorMts300C() as mts300	X.Sounder → mts300.Sounder	MTS300
*	Read<uint16_t> as Battery	new SensorMts300C() as mts300	X.Battery → mts300.Vref	MTS300
*	Read<uint16_t> as Microphone	new SensorMts300C() as mts300	X.Microphone → mts300.Microphone	MTS300
*	Read<uint16_t> as Accel_X	new SensorMts300C() as mts300	X.Accel_X → mts300.AccelX	MTS300
*	Read<uint16_t> as Accel_Y	new SensorMts300C() as mts300	X.Accel_Y → mts300.AccelY	MTS300
*	Read<uint16_t> as Mag_X	new SensorMts300C() as mts300	X.Mag_X → mts300.MagX	MTS300
*	Read<uint16_t> as Mag_Y	new SensorMts300C() as mts300	X.Mag_Y → mts300.MagY	MTS300
*	Read<uint16_t> as Temp	new TempC() as SensorTemperature	X.Temp → SensorTemperature.Read	BASICSB
*	Read<uint16_t> as Light	new PhotoC() as SensorLight	X.Light → SensorLight.Read	BASICSB
send	AMSend as SerialSendID	SerialActiveMessageC as SerialComm	X.SerialSendID → SerialComm.AMSend	
send	SplitControl as SerialControl	SerialActiveMessageC as SerialComm	X.SerialControl → SerialComm.SplitControl	
send	AMSend as SendID	new AMSenderC(ID) as SenderID	X.SendID → SenderID.AMSend	
send	SplitControl as AMControl	ActiveMessageC	X.AMControl → ActiveMessageC.SplitControl	
receive	Receive as ReceiveID	SerialActiveMessageC as SerialComm	X.ReceiveID → SerialComm.Receive	
receive	Receive as ReceiveID	new AMReceiverC(ID) as ReceiverID	X.ReceiveID → ReceiverID.Receive	
receive	SplitControl as AMControl	ActiveMessageC	X.AMControl → ActiveMessageC.SplitControl	
*	Packet	new AMSenderC(ID) as SenderID	X.Packet → SenderID.Packet	
*	Packet	new AMReceiverC(ID) as ReceiverID	X.Packet → ReceiverID.Packet	

Continued on Next Page...

Table B.3 – Continued

Service (S)	TinyOS 2.x interface (I)	TinyOS 2.x component (C)	TinyOS 2.x wiring (W)	Platforms
*	Packet	SerialActiveMessageC as SerialComm	X.Packet → SerialComm.Packet	
*	AMPacket	new AMSenderC(ID) as SenderID	X.AMPacket → SenderID.AMPacket	
*	AMPacket	new AMReceiverC(ID) as ReceiverID	X.AMPacket → ReceiverID.AMPacket	
*	AMPacket	SerialActiveMessageC as SerialComm	X.AMPacket → SerialComm.AMPacket	
*	BlockWrite	new BlockStorageC([unique("ID")])	X.BlockWrite → BlockStorageC	
*	BlockRead	new BlockStorageC([unique("ID")])	X.BlockRead → BlockStorageC	
*	LogWrite	new LogStorageC([unique("ID")], <bool>[3])	X.LogWrite → LogStorageC	
*	LogRead	new LogStorageC([unique("ID")], <bool>)	X.LogRead → LogStorageC	
*	Mount	new ConfigStorageC([unique("ID")])	X.Mount → ConfigStorageC.Mount	
*	ConfigStorage	new ConfigStorageC([unique("ID")])	X.ConfigStorage → ConfigStorageC.ConfigStorage	

Table B.3: List of interfaces (I), components (C) and wirings (W) required by a T2 program using service (S).

[2]ID is a numeric identifier representing the instance of the device.

[3]This boolean value indicates if the log is circular or not.

B.2.2. List of events in T2

Interface (I)	Events (E)
Timer	event void fired()
Read	event void readDone(error_t result, val_t value)
AMSend	event void sendDone(message_t msg, error_t error)
Receive	event message_t *receive(message_t *msg, void *payload, uint8_t len)
BlockWrite	event void writeDone(storage_addr_t addr, void *buf, storage_len_t len, error_t error)
	event void eraseDone(error_t error)
	event void syncDone(error_t error)
BlockRead	event void computeCrcDone(storage_addr_t addr, storage_len_t len, uint16_t crc error_t error)
	event void readDone(storage_addr_t addr, void *buf, storage_len_t len, error_t error)
LogWrite	event void appendDone(void *buf, storage_len_t len, bool recordsLost, error_t error)
	event void eraseDone(error_t error)
	event void syncDone(error_t error)
LogRead	event void readDone(void *buf, storage_len_t len, error_t error)
	event void seekDone(error_t error)
Mount	event void mountDone(error_t error)
ConfigStorage	event void readDone(storage_addr_t addr, void* buf, storage_len_t len, error_t error)
	event void writeDone(storage_addr_t addr, void* buf, storage_len_t len, error_t error)

Table B.4: List of events (E) to implement when a T2 application uses the interface (I)

B.3. Code generation: SN_OSAL to OS ($\nearrow_T$)

B.3.1. Code generation from SN_OSAL to T1

Function $\nearrow_T : SN\text{-}OSALI \rightarrow OI_{T1'}$ is shown in the following Table:

SN_OSAL Service ($SN\text{-}OSALI$)	**TinyOS 1.x Service** ($OI_{T1'}$)	**HW Platform**
TIMER Services		
osal_timer_start(*, SEC, ONE_SAMPLE)	call TimerID.start(TIMER_ONE_SHOT, \1 * 1000)	All
osal_timer_start(*, SEC, REPEAT_SAMPLE)	call TimerID.start(TIMER_REPEAT, \1 * 1000)	All
osal_timer_start(*, MILLISEC, ONE_SAMPLE)	call TimerID.start(TIMER_ONE_SHOT, \1)	All
osal_timer_start(*, MILLISEC, REPEAT_SAMPLE)	call TimerID.start(TIMER_REPEAT, \1)	All
osal_timer_restart(*)	call TimerX.stop() call TimerX.start(<units>,<mode>)	All
osal_timer_stop(*)	call TimerX.stop()	All
osal_gettime()	call Time.get()	All
osal_timer_ioctl(*, SET_SCALE, *)	call Clock.setNextScale(\2)	All
osal_timer_ioctl(*, SET_INTERVAL, *)	call Clock.setInterval(\2)	All
osal_timer_ioctl(*, GET_SCALE, *)	\2 = call Clock.getScale()	All
osal_timer_ioctl(*, GET_INTERVAL, *)	\2 = call Clock.getInterval()	All
DEBUGGING Services		
osal_led_on(LED_RED)	call Leds.redOn()	All
osal_led_on(LED_GREEN)	call Leds.greenOn()	All
osal_led_on(LED_YELLOW)	call Leds.yellowOn()	All
osal_led_off(LED_RED)	call Leds.redOff()	All
osal_led_off(LED_GREEN)	call Leds.greenff()	All
osal_led_off(LED_YELLOW)	call Leds.yellowOff()	All
osal_led_toggle(LED_RED)	call Leds.redToggle()	All
osal_led_toggle(LED_GREEN)	call Leds.greenToggle()	All
osal_led_toggle(LED_YELLOW)	call Leds.yellowToggle()	All

Continued on Next Page...

Table B.5 – Continued

SN_OSAL Service ($SN\text{-}OSALI$)	**TinyOS 1.x Service** ($OI_{T1'}$)[4]	**HW Platform**
osal_led_init()	call Leds.init()	All
osal_led_get()	call Leds.get()	All
osal_led_set(*)	call Leds.set(\1)	All
STORAGE AND FILE SYSTEMS Services		
osal_fs_open(*, *, O_RDONLY)	#define FILE_READ \1 call FileRead.open(\1)	Mica family
osal_fs_open(*, O_CREAT, O_WRONLY)	#define FILE_WRITE \1 call FileWrite.open(\1, FS_CREATE)	Mica family
osal_fs_open(*, O_TRUNCATE, O_WRONLY)	#define FILE_WRITE \1 call FileWrite.open(\1, FS_TRUNCATE)	Mica family
osal_fs_close(*)	#if defined(FILE_READ) call FileRead.close(); #elif defined(FILE_WRITE) call FileWrite.close(); #endif	Mica family
osal_fs_read(*,*,*)	call FileRead.read(\2, \3)	Mica family
osal_fs_write(*,*,*)	call FileWrite.append(\2, \3)	Mica family
osal_fs_opendir(*,*)	call FileDir.start()	Mica family
osal_fs_readdir(*,*)	call FileDir.readNext()	Mica family
osal_fs_closedir(*)		Mica family
SCHEDULING Services		
osal_task_create(*,*,*)	post \2() call OS_SchedServices.osal_task_create(\1, \2, \3)	All All
osal_task_create(*,*,*)	post \2() osal_task_create(\1, \2, \3)	All All
osal_task_exit()	call OS_SchedServices.osal_task_exit()	All
osal_task_current()	call OS_SchedServices.osal_task_current()	All
osal_task_pid()	call OS_SchedServices.osal_task_pid()	All
osal_task_list()	call OS_SchedServices.osal_task_list()	All

Continued on Next Page...

Table B.5 – Continued

SN_OSAL Service ($SN\text{-}OSALI$)	**TinyOS 1.x Service** ($OI_{T1'}$)[4]	**HW Platform**
osal_task_signal(OSAL_IO_READDONE, *, *)	*Generate event handler*	All
osal_task_signal(OSAL_IO_WRITEDONE, *, *)	*Generate event handler*	All
osal_task_signal(OSAL_TIMER_FIRED, *, *)	*Generate event handler*	All
osal_task_signal(OSAL_NET_SENDDONE, *, *)	*Generate event handler*	All
osal_task_signal(OSAL_NET_RECEIVE, *, *)	*Generate event handler*	All
osal_lock()	__nesc_atomic_t at = __nesc_atomic_start()	All
osal_unlock()	__nesc_atomic_stop(at)	All
SENSOR Services		
osal_io_open(temp)	call TempControl.init()	Telos family
osal_io_close(temp)	call TempControl.stop()	Telos family
osal_io_read(temp, *)	call TempControl.start() call Temp.getData()	Telos family
osal_io_open(humidity)	call HumidityControl.init()	Telos family
osal_io_close(humidity)	call HumidityControl.stop()	Telos family
osal_io_read(humidity, *)	call HumidityControl.start() call Humidity.getData()	Telos family
osal_io_open(light_PAR)		Telos family
osal_io_close(light_PAR)		Telos family
osal_io_read(light_PAR, *)	call PAR.getData()	Telos family
osal_io_open(light_TSR)		Telos family
osal_io_close(light_TSR)		Telos family
osal_io_read(light_TSR, *)	call TSR.getData()	Telos family
osal_io_open(temp)	call TempControl.init()	MTS300 & MICASB
osal_io_close(temp)	call TempControl.stop()	MTS300 & MICASB
osal_io_read(temp, *)	call TempControl.start() call Temp.getData()	MTS300 & MICASB
osal_io_open(light)	call PhotoControl.init()	MTS300 & MICASB

Continued on Next Page...

Table B.5 – Continued

SN_OSAL Service ($SN\text{-}OSALI$)	**TinyOS 1.x Service** ($OI_{T1'}$)[4]	**HW Platform**
osal_io_close(light)	call PhotoControl.stop()	MTS300 & MICASB
osal_io_read(light, *)	call PhotoControl.start()	MTS300 & MICASB
	call Light.getData()	
osal_io_open(sound)	call Sounder.init()	MTS300 & MICASB
osal_io_close(sound)	call Sounder.stop()	MTS300 & MICASB
osal_io_read(sound, *)	call Sounder.start()	MTS300 & MICASB
osal_io_open(microphone)	call MicControl.init()	MTS300 & MICASB
	call Mic.muxSel(1)	
	call Mic.gainAdjust(64)	
osal_io_close(microphone)	call MicControl.stop()	MTS300 & MICASB
osal_io_read(microphone, *)	call MicControl.start()	MTS300 & MICASB
osal_io_open(accel_x)	call AccelControl.init()	MTS300 & MICASB
osal_io_close(accel_x)	call AccelControl.stop()	MTS300 & MICASB
osal_io_read(accel_x, *)	call AccelControl.start()	MTS300 & MICASB
	call Accel_X.getData()	
osal_io_open(accel_y)	call AccelControl.init()	MTS300 & MICASB
osal_io_close(accel_y)	call AccelControl.stop()	MTS300 & MICASB
osal_io_read(accel_y, *)	call AccelControl.start()	MTS300 & MICASB
	call Accel_Y.getData()	
osal_io_open(mag_x)	call MagControl.init()	MTS300 & MICASB
osal_io_close(mag_x)	call MagControl.stop()	MTS300 & MICASB
osal_io_read(mag_x, *)	call MagControl.start()	MTS300 & MICASB
	call Mag_X.getData()	
osal_io_open(mag_y)	call MagControl.init()	MTS300 & MICASB
osal_io_close(mag_y)	call MagControl.stop()	MTS300 & MICASB
osal_io_read(mag_y, *)	call MagControl.start()	MTS300 & MICASB
	call Mag_Y.getData()	

Continued on Next Page...

Table B.5 – Continued

SN_OSAL Service ($SN\text{-}OSALI$)	**TinyOS 1.x Service** ($OI_{T1'}$)[4]	**HW Platform**
osal_io_open(temp)	call TempControl.init()	MICAWBDOT
osal_io_close(temp)	call TempControl.stop()	MICAWBDOT
osal_io_read(temp, *)	call TempControl.start()	MICAWBDOT
	call Temp.getData()	
osal_io_open(humidity)	call HumidityControl.init()	MICAWBDOT
osal_io_close(humidity)	call HumdityControl.stop()	MICAWBDOT
osal_io_read(humidity, *)	call HumidityControl.start()	MICAWBDOT
	call Humidity.getData()	
osal_io_open(light*)	call LightControl.init()	MICAWBDOT
osal_io_close(light*)	call LightControl.stop()	MICAWBDOT
osal_io_read(light_TSR, *)	call LightControl.start()	MICAWBDOT
	call LightChannel0.getData()	
osal_io_read(light_PAR, *)	call LightControl.start()	MICAWBDOT
	call LightChannel1.getData()	
osal_io_open(pressure)	call PressureControl.init()	MICAWBDOT
osal_io_close(pressure)	call PressureControl.stop()	MICAWBDOT
osal_io_read(pressure, *)	call PressureControl.start()	MICAWBDOT
	call Pressure.getData()	
osal_io_open(temp)	call TempControl.init()	BASICSB
osal_io_close(temp)	call TempControl.stop()	BASICSB
osal_io_read(temp, *)	call TempControl.start()	BASICSB
	call Temp.getData()	
osal_io_open(light*)	call PhotoControl.init()	BASICSB
osal_io_close(light*)	call PhotoControl.stop()	BASICSB
osal_io_read(light*, *)	call PhotoControl.start()	BASICSB
	call Light.getData()	
osal_io_open(battery)	call VoltageControl.init()	All

Continued on Next Page...

Table B.5 – Continued

SN_OSAL Service ($SN\text{-}OSALI$)	**TinyOS 1.x Service** ($OI_{T1'}$)[4]	**HW Platform**
osal_io_close(battery)	call VoltageControl.stop()	All
osal_io_read(battery, *)	call VoltageControl.start() call Battery.getData()	All
COMM (Net and serial) Services		
osal_net_getId()	TOS_LOCAL_ADDRESS	All
osal_net_getIdBroadcast()	TOS_BCAST_ADDR	All
osal_net_getIdSerial()	0x7d	All
osal_net_send(ADDR_SERIAL_PORT, *, *)	if (call SendID.send(msg_uart) == SUCCESS){ osal_radio_busy = TRUE;}	All

osal_net_send(ADDR_BROADCAST, *, *)	if (call SendID.send(TOS_BCAST_ADDR, \3,
	\2) == SUCCESS) {
	osal_radio_busy = TRUE;}
		All

osal_net_send(*,*,*)	if (call SendX.send(\1, \3, \2) == SUCCESS) { osal_radio_busy = TRUE;}	All
LIBRARY Services		
printf(*)	dbg(DBG_USR1,\1),	All
strcpy(*,*)	osal_copy (\1,\2)[5]	All

Table B.5: $\diagdown_T : SN\text{-}OSALI \rightarrow OI_{T1'}$

[4]The notation "\x" (with n being a number) represents the replacement of the argument by the SN-OSAL argument with number of order n.

[5]Include the "osal_utils.h" with the next function:

```
int osal_copy(int8_t dest, int8_t src)
int16_t cnt = 0;
do {dest[cnt]=src[cnt];cnt++;}while(src[cnt-1] != 0);return (cnt-1); }
```

B.3.2. Code generation from SN_OSAL to T2

Function $\searrow_T : SN\text{-}OSALI \rightarrow OI_{T2'}$ is shown in the following Table:

SN_OSAL Service ($SN\text{-}OSALI$)	**TinyOS 2.x Service** ($OI_{T2'}$)	**HW Platform**
TIMER Services		
osal_timer_start(*, SEC, ONE_SAMPLE)	call TimerID.startOneShot(\1 * 1000)	All
osal_timer_start(*, SEC, REPEAT_SAMPLE)	call TimerID.startPeriodic(\1 * 1000)	All
osal_timer_start(*, MILLISEC, ONE_SAMPLE)	call TimerID.startOneShot(\1)	All
osal_timer_start(*, MILLISEC, REPEAT_SAMPLE)	call TimerID.startPeriodic(\1)	All
osal_timer_start(*, MICROSEC, ONE_SAMPLE)	call TimerID.startOneShot(\1)	All
osal_timer_start(*, MICROSEC, REPEAT_SAMPLE)	call TimerID.startPeriodic(\1)	All
osal_timer_restart(*)	call TimerID.stop() call TimerID.start[OneShot/Periodic](<interval>)	All
osal_timer_stop(*)	call TimerID.stop()	All
osal_gettime()	call TimerID.getNow()	All
DEBUGGING Services		
osal_led_on(LED_RED)	call Leds.led0On()	All
osal_led_on(LED_GREEN)	call Leds.led1On()	All
osal_led_on(LED_YELLOW)	call Leds.led2On()	All
osal_led_off(LED_RED)	call Leds.led0Off()	All
osal_led_off(LED_GREEN)	call Leds.led1Off()	All
osal_led_off(LED_YELLOW)	call Leds.led2Off()	All
osal_led_toggle(LED_RED)	call Leds.led0Toggle()	All
osal_led_toggle(LED_GREEN)	call Leds.led1Toggle()	All
osal_led_toggle(LED_YELLOW)	call Leds.led2Toggle()	All
osal_led_init()		All
osal_led_get()	call Leds.get()	All
osal_led_set(*)	call Leds.set(\1)	All

Continued on Next Page...

Table B.6 – Continued

SN_OSAL Service ($SN\text{-}OSALI$)	**TinyOS 2.x Service** ($OI_{T2'}$)	**HW Platform**
SCHEDULING Services		
osal_task_create(*,*,*)	osal_task_create(\1, \2, \3) post \2()	All
osal_task_exit()	call OS_SchedServices.osal_task_exit()	All
osal_task_current()	call OS_SchedServices.osal_task_current()	All
osal_task_pid()	call OS_SchedServices.osal_task_pid()	All
osal_task_list()	call OS_SchedServices.osal_task_list()	All
osal_task_signal(OSAL_IO_READDONE, *, *)	*Generate event handler*	All
osal_task_signal(OSAL_IO_WRITEDONE, *, *)	*Generate event handler*	All
osal_task_signal(OSAL_TIMER_FIRED, *, *)	*Generate event handler*	All
osal_task_signal(OSAL_NET_SENDDONE, *, *)	*Generate event handler*	All
osal_task_signal(OSAL_NET_RECEIVE, *, *)	*Generate event handler*	All
osal_lock()	atomic {	All
osal_unlock()	}	All
SENSOR Services		
osal_io_open(temp)		Telos family
osal_io_close(temp)		Telos family
osal_io_read(temp, *)	call Temp.read()	Telos family
osal_io_open(humidity)		Telos family
osal_io_close(humidity)		Telos family
osal_io_read(humidity, *)	call Humidity.read()	Telos family
osal_io_open(light_PAR)		Telos family
osal_io_close(light_PAR)		Telos family
osal_io_read(light_PAR, *)	call PhotoPar.read()	Telos family
osal_io_open(light_TSR)		Telos family
osal_io_close(light_TSR)		Telos family
osal_io_read(light_TSR, *)	call PhotoTsr.read()	Telos family

Continued on Next Page...

Table B.6 – Continued

SN_OSAL Service (*SN-OSALI*)	**TinyOS 2.x Service** ($OI_{T2'}$)	**HW Platform**
osal_io_open(temp)		MTS300 & MICASB
osal_io_close(temp)		MTS300 & MICASB
osal_io_read(temp, *)	call Temp.read()	MTS300 & MICASB
osal_io_open(light*)		MTS300 & MICASB
osal_io_close(light*)		MTS300 & MICASB
osal_io_read(light*, *)	call Light.read()	MTS300 & MICASB
osal_io_open(sound)		MTS300 & MICASB
osal_io_close(sound)		MTS300 & MICASB
osal_io_read(sound, *)	call Sounder.read()	MTS300 & MICASB
osal_io_open(microphone)		MTS300 & MICASB
osal_io_close(microphone)		MTS300 & MICASB
osal_io_read(microphone, *)	call Microphone.read()	MTS300 & MICASB
osal_io_open(accel_x)		MTS300 & MICASB
osal_io_close(accel_x)		MTS300 & MICASB
osal_io_read(accel_x, *)	call Accel_X.read()	MTS300 & MICASB
osal_io_open(accel_y)		MTS300 & MICASB
osal_io_close(accel_y)		MTS300 & MICASB
osal_io_read(accel_y, *)	call Accel_Y.read()	MTS300 & MICASB
osal_io_open(mag_x)		MTS300 & MICASB
osal_io_close(mag_x)		MTS300 & MICASB
osal_io_read(mag_x, *)	call Mag_X.read()	MTS300 & MICASB
osal_io_open(magn_y)		MTS300 & MICASB
osal_io_close(mag_y)		MTS300 & MICASB
osal_io_read(mag_y, *)	call Mag_Y.read()	MTS300 & MICASB
osal_io_open(temp)		MDA100 & BASICSB
osal_io_close(temp)		MDA100 & BASICSB
osal_io_read(temp, *)	call Temp.read()	MDA100 & BASICSB

Continued on Next Page...

Table B.6 – Continued

SN_OSAL Service ($SN\text{-}OSALI$)	**TinyOS 2.x Service** ($OI_{T2'}$)	**HW Platform**
osal_io_open(light*)		MDA100 & BASICSB
osal_io_close(light*)		MDA100 & BASICSB
osal_io_read(light*, *)	call Light.read()	MDA100 & BASICSB
osal_io_open(battery)		All
osal_io_close(battery)		All
osal_io_read(battery, *)	call Battery.read()	All
COMM (Net and Serial) Services		
osal_net_getId()	TOS_NODE_ID	All
osal_net_getIdBroadcast()	AM_BROADCAST_ADDR	All
osal_net_getIdSerial()	0x7e	All
osal_net_send(ADDR_SERIAL_PORT, *, *)	if (call SerialSendID.send[ID](0x7e, \2, \3)	All
	== SUCCESS)	All
	{ osal_radio_busy = TRUE; }	
osal_net_send(ADDR_BROADCAST, *, *)	if (call SendID.send(AM_BROADCAST_ADDR,	All
	\2, \3) == SUCCESS) {	
	osal_radio_busy = TRUE; }	
osal_net_send(*, *, *)	if (call SendID.send(\1, \2, \3) == SUCCESS) {	All
	osal_radio_busy = TRUE; }	
osal_net_receive(ADDR_BROADCAST, *, *)		All
osal_net_receive(*, *, *)		All
LIBRARY Services		
printf(*)	dbg("DBG_USR1",\1),	All
strcpy(*,*)	osal_copy (\1,\2)	All

Table B.6: $\rangle_T : SN\text{-}OSALI \rightarrow OI_{T2'}$

B.3.3. Code generation from SN_OSAL to Contiki

Function $\searrow_T : SN\text{-}OSALI \rightarrow OI_{Contiki'}$ is shown in the following Table:

SN_OSAL Service ($SN\text{-}OSALI$)	**Contiki Service** ($OI_{Contiki'}$)	**HW Platforms**
TIMER services		
osal_timer_start(*, SEC, ONE_SAMPLE)	etimer_set(&tX, \1 * CLOCK_SECOND)	All
osal_timer_start(*, MILLISEC, ONE_SAMPLE)	etimer_set(&tX, \1 * CLOCK_SECOND / 1000)	All
osal_timer_start(*, SEC, REPEAT_SAMPLE)	etimer_set(&tX, \1 * CLOCK_SECOND)	All
osal_timer_start(*, MILLISEC, REPEAT_SAMPLE)	etimer_set(&tX, \1 * CLOCK_SECOND / 1000)	All
osal_timer_restart(*, *)	etimer_restart(&tX)	All
osal_timer_fired(*)	etimer_expired(&tX)	All
osal_time_init()	clock_init()	All
osal_timer_stop(*)	etimer_stop(&tX)	All
osal_gettime(*)	\1 = clock_time(void)	All
DEBUGGING services		
osal_led_on(LED_RED)	leds_on(LEDS_RED)	All
osal_led_on(LED_GREEN)	leds_on(LEDS_GREEN)	All
osal_led_on(LED_YELLOW)	leds_on(LEDS_YELLOW)	All
osal_led_off(LED_RED)	leds_off(LEDS_RED)	All
osal_led_off(LED_GREEN)	leds_off(LEDS_GREEN)	All
osal_led_off(LED_YELLOW)	leds_off(LEDS_YELLOW)	All
osal_led_toggle(LED_RED)	leds_toggle(LEDS_RED)	All
osal_led_toggle(LED_GREEN)	leds_toggle(LEDS_GREEN)	All
osal_led_toggle(LED_YELLOW)	leds_toggle(LEDS_YELLOW)	All
osal_led_init()	leds_arch_init()	All
osal_led_get()	leds_arch_get()	All
osal_led_set(*)	leds_arch_set(\1)	All
SCHEDULING Services		

Continued on Next Page...

Table B.7 – Continued

SN_OSAL Service (*SN-OSALI*)	**Service** ($OI_{Contiki'}$)	**HW Platforms**
osal_task_create(*,*,*)	osal_task_create(\1, \2, \3)	All
osal_task_exit()	PROCESS_EXIT()	All
osal_task_current()	PROCESS_CURRENT()	All
osal_task_pid()	osal_task_pid()	All
osal_task_list()	osal_task_list()	All

osal_task_signal(OSAL_IO_READDONE(*,*,*))	PROCESS_WAIT_EVENT();
	if (ev == sensors_event) && (data == &<sensor>.sensor.value) {
	// Include here \3 }
		All

osal_task_signal(OSAL_IO_WRITEDONE(*,*,*))	PROCESS_WAIT_EVENT();
	if (ev == sensors_event) && (data == &<sensor>.sensor.value) {&
	// Include here \3 }
		All

osal_task_signal(OSAL_TIMER_FIRED(*,*,*))	PROCESS_WAIT_EVENT();
	if (ev == PROCESS_EVENT_TIMER) {
	// Include here \3 }
		All

osal_task_signal(OSAL_NET_RECEIVE(*,*,*))	PROCESS_WAIT_EVENT();
	if (ev == PROCESS_EVENT_COM) {
	// Include here \3 }
		All

osal_task_signal(OSAL_NET_SENDDONE(*,*,*))	PROCESS_WAIT_EVENT();
	if (ev == sensors_event) && (data == &<sensor>.sensor.value) {
	// Include here \3 }
		All

STORING and FILE SYSTEMS services		
osal_fs_open(*, *, O_RDONLY)	cfs_open(\1, CFS_READ)	All
osal_fs_open(*, O_CREAT, O_WRONLY)	cfs_open(\1, CFS_APPEND)	All
osal_fs_open(*, O_TRUNCATE, O_WRONLY)	cfs_open(\1, CFS_WRITE)	All
osal_fs_close(*)	cfs_close(\1)	All
osal_fs_read(*,*,*)	cfs_read(\1, \2, \3)	All
osal_fs_write(*,*,*)	cfs_write(\1, \2, \3)	All

Continued on Next Page...

Table B.7 – Continued

SN_OSAL Service (*SN-OSALI*)	**Service** ($OI_{Contiki'}$)	**HW Platforms**
osal_fs_opendir(*,*)	cfs_opendir(\1, \2)	All
osal_fs_readdir(*,*)	cfs_readdir(\1, \2)	All
osal_fs_closedir(*)	cfs_closedir(\1)	All
SENSOR Services		
osal_io_open(temp)	temperature_sensor.activate()	All
osal_io_close(temp)	temperature_sensor.deactivate()	All
osal_io_read(temp, *)	data=temperature_sensor.value(0)	All
osal_io_open(vib)	vib_sensor.activate()	All
osal_io_close(vib)	vib_sensor.deactivate()	All
osal_io_read(vib, *)	data=vib_sensor.value(0)	All
osal_io_open(pir)	pir_sensor.activate()	All
osal_io_close(pir)	pir_sensor.deactivate()	All
osal_io_read(pir, *)	data=pir_sensor.value(0)	All
osal_io_open(light_PAR)	sensors_light_init()	All
osal_io_close(light_PAR)		All
osal_io_read(light_PAR, *)	data=sensors_light1()	All
osal_io_open(light_TSR)	sensors_light_init()	All
osal_io_close(light_TSR)		All
osal_io_read(light_TSR, *)	data=sensors_light2()	All
osal_io_open(light)	sensors_light_init()	All
osal_io_close(light)		All
osal_io_read(light, *)	data=sensors_light1()	All
osal_io_open(sound)	sound_sensor.activate()	All
osal_io_close(sound)	sound_sensor.deactivate()	All
osal_io_read(sound, *)	data=sound_sensor.value(0)	All
osal_io_open(battery)	battery_sensor.activate()	All
osal_io_close(battery)	battery_sensor.deactivate()	All

Continued on Next Page...

Table B.7 – Continued

SN_OSAL Service ($SN\text{-}OSALI$)	**Service** ($OI_{Contiki'}$)	**HW Platforms**
osal_io_read(battery, *)	data=battery_sensor.value(0)	All
osal_io_open(humidity)		All
osal_io_close(humidity)		All
osal_io_read(humidity, *)	data=sht11_humidity()	All
COMM (Network and serial) Services		
osal_net_getId()	(unsigned short) node_id	All

osal_net_send(*, *, *)	rimebuf_clear();
	rimebuf_copyfrom(\2, \3);
	addr.u8[0]=addr.u8[1]=(\1);
	Y[6]_send(*X*_conn,&addr);
		All

osal_net_send(ADDR_SERIAL_PORT, *, *)	#include "dev/uart1.h"
	buffer=(unsigned char *)&osal_pkt;
	uart1_init(BAUD2UBR(115200));
	for (i=0;i<sizeof(osal_pkt);i++)
	uart1_writeb((unsigned char *)buffer[i]);
		SKY

osal_net_send(ADDR_SERIAL_PORT, *, *)	#include "dev/rs232.h"
	rs232_print(&osal_pkt);
		ESB

Table B.7: $\searrow_T : SN\text{-}OSALI \rightarrow OI_{Contiki'}$

[6] *Y* corresponds to the Rime protocol name used, and *X* to the application name.

Appendix C

OSAL functions description

C.1. Core functions

C.1.1. I/O primitives

C.1.1.1. OSAL_IO_OPEN(1) OSAL core functions v0.1 OSAL_IO_OPEN(1)

NAME

`osal_io_open` – Initialize a sensor

SYNOPSIS

```
int8_t osal_io_open (unsigned char device);
```

DESCRIPTION

The open function is a function of control defined on a sensor/actuator device intended to initialize such device. Most cases consists of providing energy to the hardware component involved. A reading operation cannot be initiated while there is a pending one. The device to initialize is identified using the `device` parameter, which is a constant that can take one of the following values:

`TEMP` – temperature sensor.

`HUMIDITY` – humidity sensor.

`LIGHT` – light sensor.

`ACOUSTIC` – acoustic sensor.

`SOUNDER` – sounder sensor.

`MICROPHONE` – microphone.

`PRESSURE` – pressure sensor.

`VIBRATION` – vibration sensor.

`ACCEL_X` – Accelerometer sensor, X axis.

`ACCEL_Y` – Accelerometer sensor, Y axis.

`MAG_X` – Magnetometer sensor, X axis.

`MAG_Y` – Magnetometer sensor, Y axis.

In the current version actuators were not implemented.

EXIT STATUS

`osal_io_open` returns a zero value if the device was successfully initialized. Non zero is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_io_close`(1), `osal_io_read`(1)

C.1.1.2. OSAL_IO_CLOSE(1) OSAL core functions v0.1 OSAL_IO_CLOSE(1)

NAME

`osal_io_close` – Close a sensor

SYNOPSIS

```
int8_t osal_io_close (int8_t desc);
```

DESCRIPTION

The close function stops a sensor device specified by the `device` parameter. No further operations can be carried out on these device.

EXIT STATUS

`osal_io_close` returns a zero value if the operation had success. Non zero is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_io_open`(1), `osal_io_read`(1)

C.1.1.3. OSAL_IO_READ(1) OSAL core functions v0.1 OSAL_IO_READ(1)

NAME

`osal_io_read` – Read a sensor

SYNOPSIS

```
int8_t osal_io_read (int8_t desc, uint16_t data);
```

DESCRIPTION

The read function initiates the reading on a device sensor specified by the `device` parameter. This operation includes the sampling over a sensor and the ADC conversion of data. The operation is asynchronous and split-phase: firstly, this functions is required to initiate the sampling. Secondly, the `OSAL_READDONE` event is signaled to indicate when data is available, and subsequently, the `data` parameter is loaded with the resulting 16-bit value.

EXIT STATUS

`osal_io_read` returns a zero value if the operation had success. Non zero is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_io_open`(1), `osal_io_close`(1)

C.1.2. Clock & energy saving primitives

C.1.2.1. OSAL_TIMER_START(1) OSAL core functions v0.1 OSAL_TIMER_START(1)

NAME

`osal_timer_start` – Start a timer

SYNOPSIS

```
int8_t osal_timer_start (int8_t num, int8_t granularity, uint8_t frequency);
```

DESCRIPTION

This function is used to set a timer event which will be signaled after a given time. The number of timers available depends on the clock device. It takes the following parameters:

`num` indicates the time after which the event will be signaled.

`granularity` represents the units of the timer. It can take one of the following values:

`SEC` – seconds.

`MILLISEC` – milliseconds.

`MICROSEC` – microseconds.

`frequency` can take one of the following values:

`ONE_SAMPLE` – timer expiring one single time.

`REPEAT_SAMPLE` – a repetitive timer which will initiate the timer again after the last fire.

When `num granularity` has expired the `OSAL_TIMERFIRED` event will be signaled.

EXIT STATUS

`osal_timer_start` returns a non-zero timer descriptor if the operation had success. This timer descriptor must be used in subsequent operations on the same timer. -1 is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_timer_stop(1), osal_timer_restart(1), osal_timer_ioctl(1)

C.1.2.2. OSAL_TIMER_STOP(1) OSAL core functions v0.1 OSAL_TIMER_STOP(1)

NAME

`osal_timer_stop` – Stop a timer

SYNOPSIS

```
int8_t osal_timer_stop (int8_t desc);
```

DESCRIPTION

This function stops a timer previously initiated by the `osal_timer_start` primitive.

`desc` indicates the timer descriptor which will be stopped.

EXIT STATUS

`osal_timer_stop` returns zero if the timer was stopped. Non zero value is returned in case of failure, for instance, if the timer descriptor does not match to one initiated timer.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_timer_start(1), osal_timer_restart(1), osal_timer_ioctl(1)

C.1.2.3. OSAL_TIMER_RESTART(1) OSAL core functions v0.1 OSAL_TIMER_RESTART(1)

NAME

`osal_timer_restart` – Restart a timer

SYNOPSIS

```
int8_t osal_timer_restart (int8_t desc);
```

DESCRIPTION

This function restarts a timer previously initiated by `osal_timer_start` primitive. The timer is restarted from the current point in time, and it keeps the same configuration.

`desc` indicates the timer descriptor which will be restarted.

EXIT STATUS

`osal_timer_restart` returns zero if the timer was restarted. Non zero value is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_timer_start(1), osal_timer_stop(1), osal_timer_ioctl(1)

C.1.2.4. OSAL_TIMER_IOCTL(1) OSAL core functions v0.1 OSAL_TIMER_IOCTL(1)

NAME

`osal_timer_ioctl` – Configure the hardware clock settings.

SYNOPSIS

```
int8_t osal_timer_ioctl(int8_t cmd, int16_t value);
```

DESCRIPTION

This function is intended to configure the settings of a hardware clock device. Three settings were considered: `scale`, represents the number of ticks per second; `interval` represents the number of ticks per clock firing; `counter` indicates the hardware clock counter. This function takes two arguments:

`cmd` indicates the action to be done. It can take one of the following values:

`GET_SCALE` obtain the current value for `scale`.

`GET_INTERVAL` obtain the current value for `interval`.

`GET_COUNTER` obtain the current value for `counter`.

`SET_SCALE` fixes `scale` with `value` argument.

`SET_INTERVAL` fixes `interval` with `value` argument.

`SET_COUNTER` fixes `counter` with `value` argument.

`value` indicates a value when the previous action is of type `set`. No value is required in other case.

EXIT STATUS

`osal_timer_ioctl` returns zero if the timer was restarted. Non zero value is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_timer_start(1), osal_timer_stop(1), osal_timer_restart(1)

C.1.2.5. OSAL_DEVICE_ON(1) OSAL core functions v0.1 OSAL_DEVICE_ON(1)

NAME

`osal_device_on` – Power on a device

SYNOPSIS

```
uint8_t osal_device_on (uint8_t device);
```

DESCRIPTION

This function is intended to power on a hardware device. The function receives one argument: `device` represents the device to be powered on (e.g. radio).

EXIT STATUS

`osal_device_on` returns zero if the component was powered on. A value bigger than 0 is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_device_off(1)

C.1.2.6. OSAL_DEVICE_OFF(1) OSAL core functions v0.1 OSAL_DEVICE_OFF(1)

NAME

`osal_device_off` – Power off a device

SYNOPSIS

```
uint8_t osal_device_off (uint8_t device);
```

DESCRIPTION

This function is intended to power off a hardware device. The function receives one argument: `device` represents the device to be powered off (e.g. radio).

EXIT STATUS

`osal_device_off` returns zero if the component was powered off. A value bigger than 0 is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_device_on(1)

C.1.3. Communication primitives

C.1.3.1. OSAL_NET_SEND(1) OSAL core functions v0.1 OSAL_NET_SEND(1)

NAME

`osal_net_send` – Send a message

SYNOPSIS

```
int8_t osal_net_send (uint8_t addr, void *data, uint8_t length);
```

DESCRIPTION

This function sends a message to a specified address. The address can be: the local address of a node, the broadcast address (0xFF), or the address of the serial port. The high-level message to send must be defined by the application, which must customize the network structure called `osal_comm_message`, including the definition of required fields. This structure will be encapsulated as data into the low-level package. The maximum message size (MMS) must respect the implementation of the network protocol to be used, and therefore, must be limited.

The function takes three arguments:

`addr` indicates the address for the final receptor of the message.

`data` is a pointer to the message structure.

`length` indicates the number of bytes occupied by the message structure. This operation is also split-phase: the send will be only completed when the `OSAL_SENDDONE` event is signaled.

EXIT STATUS

`osal_net_send` returns zero if the operation was successful. Non zero value is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_net_getId(1), osal_net_getBroadcast(1), osal_net_getIdSerial(1)

C.1.3.2. OSAL_NET_GETID(1) OSAL core functions v0.1 OSAL_NET_GETID(1)

NAME

`osal_net_getId` – Get the local address

SYNOPSIS

```
uint8_t osal_net_getId ();
```

DESCRIPTION

This function obtains the local address of a sensor node. The address ranges between 0x00 (for base station) and 0xfe (other nodes).

EXIT STATUS

`osal_net_getId` returns always the address of the node.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_net_send(1), osal_net_getBroadcast(1), osal_net_getIdSerial(1)

C.1.3.3. OSAL_NET_GETIDBROADCAST(1) OSAL core functions v0.1 OSAL_NET_GETIDBROADCAST(1)

NAME

`osal_net_getIdBroadcast` – Get the broadcast address

SYNOPSIS

```
int16_t osal_net_getIdBroadcast ();
```

DESCRIPTION

This function obtains the broadcast address of the network, typically 0xff.

EXIT STATUS

`osal_net_getIdBroadcast` returns the broadcast address.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_net_send(1), osal_net_getBroadcast(1), osal_net_getIdSerial(1)

C.1.3.4. OSAL_NET_GETIDSERIAL(1) OSAL core functions v0.1 OSAL_NET_GETIDSERIAL(1)

NAME

`osal_net_getIdSerial` – Get the serial address.

SYNOPSIS

```
uint8_t osal_net_getIdSerial ();
```

DESCRIPTION

This function obtains the serial address of the node, typically 0x7e. Note that this function makes sense only when the invoker node is a gateway, which has a serial port to forward messages to PC.

EXIT STATUS

`osal_net_getIdSerial` returns the broadcast address. Non zero value is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_net_send(1), osal_net_getBroadcast(1), osal_net_getId(1)

C.1.4. Storage primitives

C.1.4.1. OSAL_FS_OPEN(1) OSAL core functions v0.1 OSAL_FS_OPEN(1)

NAME

`osal_fs_open` – Open a file

SYNOPSIS

```
int8_t osal_fs_open (char *filename, uint8_t mode);
```

DESCRIPTION

The open function uses the lookup operation for locating the file `inode`. The file path lookup is an internal operation employed by file open. It works in the following way: it traverses a height level of the tree looking for a components among the right-brothers of a file/directory. If it does not find it the lookup fails. If a node is found, the lookup descends to the child and the search continues in the same way. The number of files in a mote is expectedly low, tens of files and directories, therefore, the cost of this simple approach is low. If it is found, it retrieves the `inode` and reserves the first free available entry in the open file table of the corresponding partition and returns a reference to it (file descriptor). This file descriptor will be used for identifying the file in subsequent accesses. If the file does not exist, it is created. When a file is created adds a brother node to the right-most brother of a tree level, therefore it implies the modification of an `inode` and the allocation of a free `inode`.

`filename` is the name of the file.

`mode` takes one of the following values:

`WRITE`

`READ`

EXIT STATUS

`osal_fs_open` returns a non-zero file descriptor if the operation had success. This file descriptor must be used in subsequent operations on the same file. -1 is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_fs_close(1), osal_fs_write(1), osal_fs_read(1), osal_fs_lseek(1), osal_fs_rename(1), osal_fs_stat(1), osal_fs_delete(1),

C.1.4.2. OSAL_FS_CLOSE(1) OSAL core functions v0.1 OSAL_FS_CLOSE(1)

NAME

osal_fs_close – Close a file

SYNOPSIS

```
int8_t osal_fs_close (uint8_t fd);
```

DESCRIPTION

The close function frees the entry in the open file table associated with the file.

fd is the file descriptor identifying the file to close.

EXIT STATUS

osal_fs_close returns a zero exist status if it succeeds to close the file. Non zero is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_fs_open(1), osal_fs_write(1), osal_fs_read(1), osal_fs_lseek(1), osal_fs_rename(1), osal_fs_stat(1), osal_fs_delete(1)

C.1.4.3. OSAL_FS_WRITE(1) OSAL core functions v0.1 OSAL_FS_WRITE(1)

NAME

osal_fs_write – Append data to a file

SYNOPSIS

```
int8_t osal_fs_write (uint8_t fd, char *buffer, int8_t length);
```

DESCRIPTION

The write operation appends data to the end of a file. The modification is done in a small buffer cache in RAM and it is committed to the flash either when a page is completely written or when the RAM is full. The first case tries to avoid that a page is committed to flash several times for small writes.

`fd` is the file descriptor identifying the file to close.

`buffer` contains the data to be written.

`length` is the amount of data in bytes to append.

EXIT STATUS

`osal_fs_write` returns the number of bytes written to a file. -1 is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_fs_open`(1), `osal_fs_close`(1), `osal_fs_read`(1), `osal_fs_lseek`(1), `osal_fs_rename`(1), `osal_fs_stat`(1), `osal_fs_delete`(1)

C.1.4.4. OSAL_FS_READ(1) OSAL core functions v0.1 OSAL_FS_READ(1)

NAME

`osal_fs_read` – Read from a file

SYNOPSIS

```
int8_t osal_fs_read (uint8_t fd, char *buffer, int8_t length);
```

DESCRIPTION

This operation reads the data from the flash memory to an application buffer. If the data is already in the small buffer cache, it is copied to the application buffer from there.

fd is the file descriptor identifying the file to close.

buffer is the application buffer where data will be recovered.

length is the amount of data in bytes to read.

EXIT STATUS

`osal_fs_read` returns the number of bytes read from a file. -1 is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_fs_open(1), **osal_fs_close**(1), **osal_fs_write**(1),**osal_fs_lseek**(1), **osal_fs_rename**(1), **osal_fs_stat**(1), **osal_fs_delete**(1)

C.1.4.5. OSAL_FS_RENAME(1) OSAL core functions v0.1 OSAL_FS_RENAME(1)

NAME

`osal_fs_rename` – Rename a file

SYNOPSIS

```
int8_t osal_fs_rename(char *oldname, char *newname);
```

DESCRIPTION

This operation renames an existing file with a new name. The `oldname` parameter is looked for in the name space. If it is found then this name is replaced by the `newname` parameter.

`oldname` is the current name of the file.

`newname` is the new name of the file.

EXIT STATUS

`osal_fs_rename` returns a zero exist status if it succeeds to change the file name. -1 is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_fs_open`(1), `osal_fs_close`(1), `osal_fs_write`(1), `osal_fs_read`(1), `osal_fs_lseek`(1), `osal_fs_delete`(1), `osal_fs_stat`(1)

C.1.4.6. OSAL_FS_LSEEK(1) OSAL core functions v0.1 OSAL_FS_LSEEK(1)

NAME

`osal_fs_lseek` – Update the offset of a file

SYNOPSIS

```
int8_t osal_fs_lseek(uint8_t fd, uint32_t ptr);
```

DESCRIPTION

This operation modifies the access pointer of an open file to an offset specified.

`fd` is the file descriptor identifying the file.

`ptr` is a 32-bit value specifying the new position to access data inside of the file.

EXIT STATUS

`osal_fs_lseek` returns a zero if the position `ptr` is lower than the size of file. Non zero is returned in other case.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_fs_open`(1), `osal_fs_close`(1), `osal_fs_write`(1), `osal_fs_read`(1), `osal_fs_delete`(1), `osal_fs_rename`(1), `osal_fs_stat`(1)

C.1.4.7. OSAL_FS_STAT(1) OSAL core functions v0.1 OSAL_FS_STAT(1)

NAME

`osal_fs_stat` – Obtain metadata of a file

SYNOPSIS

```
int8_t osal_fs_stat(uint8_t fd, struct inode *inode);
```

DESCRIPTION

This operation obtains the metadata associated to a given file. Metadata is represented by the inode structure in the following way:

```
struct inode{
uint8_t type :  1;
uint8_t firstSegmentId :  6;
uint8_t brotherInode;
uint8_t childInode;
}
```

where type indicates if the inode corresponds to a file or to a directory. The firstSegmentId identifies the first segment of the current file and describes the next data:

```
struct segment{
uint8_t writeCounter :  14;
uint8_t firstPagePtr :  5;
uint8_t nextSegmentID : 6;
}
```

The writeCounter indicates the number of times the pages of this segment have been written. The firstPagePtr is a circular pointer: when it reaches the end of the segment, the pointer is assigned again at the beginning of the segment and the write counter is incremented. In case a file to which the current segment was assigned is deleted, the current pointer value is kept for a future file assignment. In this way, a perfect wear leveling may be achieved inside each segment for the write streams such as the files generated by queries and materialization points. The nextSegmentID is used as an additional level of indirection for the files that require more than one segment. The brotherInode and childInode are two inodes corresponding to the brother and child nodes in the name space tree. Besides of that information, the name of the file is returned.

fd is the file descriptor identifying the file.

inode is a pointer to a inode structure defined as above indicated.

EXIT STATUS

osal_fs_stat returns a zero if the given file is found in the file system. Non zero is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

osal_fs_open(1), osal_fs_close(1), osal_fs_write(1), osal_fs_read(1), osal_fs_lseek(1), osal_fs_rename(1), osal_fs_delete(1)

C.1.4.8. OSAL_FS_DELETE(1) OSAL core functions v0.1 OSAL_FS_DELETE(1)

NAME

`osal_fs_delete` – Delete a file

SYNOPSIS

```
int8_t osal_fs_delete (uint8_t fd);
```

DESCRIPTION

File deletion updates the `inode` of the brother and father of the current directory and frees the current `inode`.

`fd` is the file descriptor identifying the file.

EXIT STATUS

`osal_fs_delete` returns a zero if the file is deleted. Non zero is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_fs_open`(1), `osal_fs_close`(1), `osal_fs_write`(1), `osal_fs_read`(1), `osal_fs_lseek`(1), `osal_fs_rename`(1), `osal_fs_stat`(1)

C.1.5. LEDs primitives

C.1.5.1. OSAL_LED_ON(1) OSAL core functions v0.1 OSAL_LED_ON(1)

NAME

`osal_led_on` – Turn on the led

SYNOPSIS

```
void osal_led_on(int8_t led);
```

DESCRIPTION

This function turns on the led device specified by the `led` argument. In general, motes have three led devices: red, green and yellow. Combinations of the three leds can provide until eight possible values, which can help to programmers to debug their programs. The `led` argument is a constant identifying the device, which can take one of the following values:

`LED_RED` identifies the red led.

`LED_YELLOW` identifies the yellow led.

`LED_GREEN` identifies the green led.

EXIT STATUS

No value is returned by this function.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_led_off`(1), `osal_led_toggle`(1)

C.1.5.2. OSAL_LED_OFF(1) OSAL core functions v0.1 OSAL_LED_OFF(1)

NAME

osal_led_off – Turn off the led

SYNOPSIS

void osal_led_off(int8_t led);

DESCRIPTION

This function turns off the led device specified by the `led` argument.

EXIT STATUS

No value is returned by this function.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_led_on`(1), `osal_led_toggle`(1)

C.1.5.3. OSAL_LED_TOGGLE(1) OSAL core functions v0.1 OSAL_LED_TOGGLE(1)

NAME

`osal_led_toggle` – Toggle the led

SYNOPSIS

```
void osal_led_toggle(int8_t led);
```

DESCRIPTION

This function toggles the led device specified by the `led` argument.

EXIT STATUS

No value is returned by this function.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_led_on`(1), `osal_led_off`(1)

C.2. Library functions

C.2.1. Task primitives

C.2.1.1. OSAL_TASK_CREATE(1) Library functions v0.1 OSAL_TASK_CREATE(1)

NAME

`osal_task_create` – Schedule an OSAL task

SYNOPSIS

```
#include "OS_SchedServices.h"
int8_t osal_task_create(char *name, void * (*start_routine)(void *), void *args);
```

DESCRIPTION

This library provides some useful information about the high-level process created in the OSAL applications. The abstractions defined in this library register the OSAL processes, and track the processes scheduled by the applications. The `osal_task_create` function schedules a new OSAL process to be lately executed. For every new OSAL process an structure of type `osal_task_attr` is allocated at the end of the process queue. OSAL fulfills that structure using the information given as arguments:

`name` the logical name of the process.

`start_routine` is the function to be executed by the process.

`args` is a pointer to an structure including the arguments for the process. The structure is defined as follows:

```
struct osal_task_attr{
int osal_pid;
char osal_task_name[64];
void *osal_task;
struct osal_task_attr *osal_next;
} osal_task;
```

where:

`osal_tid` is an univocal process identifier for OSAL process.

`osal_task_name` is the logical name of the process.

`osal_task` is a pointer to the function implemented by the process.

`osal_next` is a pointer to the next OSAL process in the queue.

An OSAL process is mapped to the equivalent execution entity in the underlying operating system, for which the application will be translated. Subsequently, its execution will depend on the policies implemented at the operating system level.

EXIT STATUS

osal_task_create returns a zero if an structure could be allocated by the process. Non zero is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_task_current`(1), `osal_task_exit`(1), `osal_task_tid`(1), `osal_task_list`(1), `osal_task_signal`(1)

C.2.1.2. OSAL_TASK_CURRENT(1) Library functions v0.1 OSAL_TASK_CURRENT(1)

NAME

`osal_task_current` – Obtain the metadata of an OSAL process

SYNOPSIS

```
#include "OS_SchedServices.h"
osal_task* osal_task_current();
```

DESCRIPTION

The `osal_task_current` function allows to obtain the information registered by OSAL about the invoker process. In particular, it returns a pointer to its `osal_task` structure. Application has access to data through the pointer returned.

EXIT STATUS

`osal_task_current` returns a pointer to the `osal_task` structure of the invoker process.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_task_create`(1), `osal_task_exit`(1), `osal_task_tid`(1), `osal_task_list`(1), `osal_task_signal`(1)

C.2.1.3. OSAL_TASK_EXIT(1) Library functions v0.1 OSAL_TASK_EXIT(1)

NAME

`osal_task_exit` – Remove an OSAL process from the queue

SYNOPSIS

```
#include "OS_SchedServices.h"
int8_t osal_task_exit();
```

DESCRIPTION

This function removes the invoker process from the process queue.

EXIT STATUS

`osal_task_exit` returns a zero if the process was removed. Non zero is returned in case of failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_task_create`(1), `osal_task_current`(1), `osal_task_tid`(1), `osal_task_list`(1), `osal_task_signal`(1)

C.2.1.4. OSAL_TASK_TID(1) Library functions v0.1 OSAL_TASK_TID(1)

NAME

`osal_task_tid` – Obtain the process identifier

SYNOPSIS

```
#include "OS_SchedServices.h"
int8_t osal_task_tid();
```

DESCRIPTION

The `osal_task_tid` function returns the process identifier of the invoker process. The value must range between 0 and the number of scheduled process less one.

EXIT STATUS

`osal_task_tid` returns the task identifier value.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_task_create`(1), `osal_task_current`(1), `osal_task_exit`(1), `osal_task_list`(1), `osal_task_signal`(1)

C.2.1.5. OSAL_TASK_LIST(1) Library functions v0.1 OSAL_TASK_LIST(1)

NAME

`osal_task_list` – print a list of the scheduled process.

SYNOPSIS

```
#include "OS_SchedServices.h"
int8_t osal_task_list();
```

DESCRIPTION

The `osal_task_list` prints a list of the scheduled process and the related information. Note that this function uses the display for printing, by means that it is useful for applications debugging. It uses the `printf` function to print data.

EXIT STATUS

`osal_task_list` returns always zero.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_task_create`(1), `osal_task_current`(1), `osal_task_exit`(1), `osal_task_tid`(1), `osal_task_signal`(1)

C.2.1.6. OSAL_TASK_SIGNAL(1) Library functions v0.1 OSAL_TASK_SIGNAL(1)

NAME

`osal_task_signal` – Establishes a handler event

SYNOPSIS

```
#include "OS_SchedServices.h"
uint8_t osal_task_signal(int event, int desc, osal_sighandler_t handler);
```

DESCRIPTION

The `osal_task_signal` function installs a new handler event for the event identified by `event`. Due to different instances of devices (e.g timers) could receive such event, the identifier of the device `desc` must be also provided as argument. The event handler is established by the third argument called `handler`, which specifies the function name to be executed when the signal is received.

EXIT STATUS

`osal_task_signal` returns zero if the event handler was correctly installed. A value bigger than zero is returned if failure.

AUTHOR

Soledad Escolar (soledad.escolar (at) gmail.com)

SEE ALSO

`osal_task_create`(1), `osal_task_current`(1), `osal_task_exit`(1), `osal_task_tid`(1), `osal_task_list`(1)

Bibliography

[ABC+04] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 582–589, Washington, DC, USA, 2004. IEEE Computer Society.

[Aco02] Knowles Acoustic. SiSonic design guide. http://www.knowlesacoustic.com, 2002.

[AS08] Mohammed Aal Salem. *A Real-Time Communication Framework for Wireless Sensor Networks*. PhD thesis, The School of Information Technologies. The University of Sydney, 2008.

[ASSC02] Lan F. Akyildiz, Welljan Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks, 2002.

[Atm08] Atmel. Atmel 8-bit AVR microcontroller Datasheet. http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf, 2008.

[BA06] Kenneth C. Barr and Krste Asanović. Energy-aware lossless data compression. *ACM Trans. Comput. Syst.*, 24(3):250–291, 2006.

[Bac59] John W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *IFIP Congress*, pages 125–131, 1959.

[BBB04] J. Burrell, T. Brooke, and R. Beckwith. Vineyard computing: sensor networks in agricultural production. *Pervasive Computing*, 3(1):38–45, January-March 2004.

[BC06] Paolo Bellavista and Antonio Corradi. *The Handbook of Mobile Middleware*. Auerbach Publications, Boston, MA, USA, 2006.

[BCD+05] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *MONET*, 10(4):563–579, 2005.

[BG04] D. Estrin B. Greenstein, E. Kohler. A sensor network application construction kit (SNACK). In *Proceedings of the Second ACM Conference on Embedded Networked Sensing Systems (SenSys)*, pages 200–209, 2004.

[BHM+00] Barry W. Boehm, Ellis Horowitz, Ray Madachy, Donald Reifer, Bradford K. Clark, Bert Steece, Winsor A. Brown, Sunita Chulani, and Chris Abts. *Software Cost Estimation with COCOMO II*. Prentice Hall PTR, January 2000.

[BHS03] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 187–200, New York, NY, USA, 2003. ACM.

[Bou09] Azzedine Boukerche. *Algorithms and Protocols for Wireless Sensor Networks*, volume ISBN 978-0-471-79813-2 of *Wiley Series on Parallel and Distributed Computing*. John Wiley & Sons., 2009.

[BP08] Amol B. Bakshi and Viktor K. Prasanna. *Architecture-Independent Programming for Wireless Sensor Networks*, volume ISBN 978-0-471-77889-9 of *Wiley Series on Parallel and Distributed Computing*. John Wiley & Sons., 2008.

[BSW06] Nicolas Burri, Roland Schuler, and Roger Wattenhofer. YETI: A TinyOS plug-in for Eclipse. In *ACM Workshop on Real-World Wireless Sensor Networks (REALWSN), Uppsala, Sweden*, June 2006.

[CA06] Qing Cao and Tarek Abdelzaher. LiteOS: a lightweight operating system for C++ software development in sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 361–362, New York, NY, USA, 2006. ACM.

[CDE+05] David Culler, Prabal Dutta, Cheng Tien Ee, Rodrigo Fonseca, Jonathan Hui, Philip Levis, Joseph Polastre, Scott Shenker, Ion Stoica, Gilman Tolle, and Jerry Zhao. Towards a sensor network architecture: Lowering the waistline. In *In Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[CGG+05] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. TinyLIME: Bridging mobile and sensor networks through middleware. In *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 61–72, Washington, DC, USA, 2005. IEEE Computer Society.

[CHAa] Atmel AT45DB011 Serial DataFlash. http://www.datasheetcatalog.com/datasheets_pdf/A/T/4/5/AT45DB.shtml.

[CHAb] CC1000 Single Chip Very Low Power RF Transceiver. http://focus.ti.com/lit/ds/symlink/cc1000.pdf.

[CHAc] CC2400 2.4GHz Low-Power RF Transceiver. http://focus.ti.com/lit/ds/symlink/cc2400.pdf.

[CHAd] Crossbow Technology Inc. *http://www.xbow.com/*.

[CHAe] M25P40 Serial Flash Memory. http://www.datasheetcatalog.org/datasheet/stmicroelectronics/7737.pdf.

[CHAf] nRF2401 Radio Transceiver Data Sheet. http://www.nvlsi.no/.

[CHAg] Sentilla. *http://www.sentilla.com/*.

[CHAh] Sustainable Bridges. *http://www.sustainablebridges.net/*.

[CHAi] TR1000 hybrid transceiver.
http://www.wirelessis.com/products/data/tr1000.pdf.

[CHAj] ZigBee Alliance. *http://www.zigbee.org*.

[CHA01] Model Driven Architecture. documentormsc/2001-07-01, Architecture Board ORMSC, 2001.

[CHA03] 10 Emerging Technologies that Will Change the World. *MIT Technology Review*, February 2003.

[CHA08] Extensible markup language (XML) 1.0.
http://www.w3.org/TR/REC-xml/, 2008.

[Cho57] Noam Chomsky. *Syntactic structures*. Mouton, The Hague, 1957.

[CK03] Chee-Yee Chong and S.P. Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, Aug. 2003.

[Cle88] J. Craig Cleaveland. Building application generators. *IEEE Software*, 5:25–33, 1988.

[CLZ05] Elaine Cheong, Edward A. Lee, and Yang Zhao. VIPTOS: a graphical development and simulation environment for Tinyos-based wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, New York, NY, USA, 2005. ACM.

[Con63] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963.

[CRH99] Erik Cota-Robles and James P. Held. A comparison of Windows driver model latency performance on Windows NT and Windows 98. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 159–172, Berkeley, CA, USA, 1999. USENIX Association.

[CSAH08] Q. Cao, J. A. Stankovic, T. F. Abdelzaher, and T. He. LiteOS, A Unix-like operating system and programming platform for wireless sensor networks. In *Information Processing in Sensor Networks(IPSN/SPOTS)*, St. Loius, MO, USA, 2008.

[DGV] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In First IEEE Workshop on Embedded Networked Sensors, 2004.

[DNH04] Hui Dai, Michael Neufeld, and Richard Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 176–187, New York, NY, USA, 2004. ACM.

[DRD+07] H. Diall, K. Raja, I. Daskalopoulos, S. Hailes, G. Roussos, and C. Torfs, T.and Van Hoof. Sensor Cube: A modular, ultra-compact, power aware platform for sensor networks. 2007.

[DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[Dun03] Adam Dunkels. Full tcp/ip for 8-bit architectures. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98, New York, NY, USA, 2003. ACM.

[Dun07a] Adam Dunkels. Porting Contiki crash course. www.sics.se/~adam/contiki-workshop-2007/workshop07porting.ppt, March 2007.

[Dun07b] Adam Dunkels. *Programming Memory-Constrained Networked Embedded Systems*. PhD thesis, 2007.

[EBMP+05] L. Evers, M. J. J. Bijl, M. Marin-Perianu, R. S. Marin-Perianu, and P. J. M. Havinga. Wireless sensor networks and beyond: A case study on transport and logistics. Technical Report TR-CTIT-05-26, University of Twente, Enschede, June 2005.

[ECG+06] Soledad Escolar, Jesús Carretero, Félix García, Florin Isaila, and Javier Fernandez. Acabando con los desarrollos ad-hoc en Wireless Sensor Networks. In *XVII Jornadas de Paralelismo*, Albacete, Spain, September 2006.

[ECGS92] Thorsten Von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[ECIC07] Soledad Escolar, Jesús Carretero, Florin Isaila, and Félix García Carballeira. A driver model based on Linux for TinyOS. In *SIES, Second IEEE Symposium on Industrial Embedded Systems*, pages 361–364, Lisboa, Portugal, 2007. IEEE.

[ECIG06] S. Escolar, J. Carretero, F. Isaila, and F. García. Deconstructing the Wireless Sensor Networks Architecture. In *SIES '06: IEEE First Symposium on Industrial Embedded Systems*, Niza, France, October 2006.

[ECIL08] Soledad Escolar, Jesús Carretero, Florin Isaila, and Stefano Lama. A lightweight storage system for sensor nodes. In Hamid R. Arabnia and Youngsong Mun, editors, *PDPTA*, pages 638–644. CSREA Press, 2008.

[ECIT08] Soledad Escolar, Jesús Carretero, Florin Isaila, and Giacomo Tartari. A MDA-based development framework for sensor network applications. In *DCOSS*, 2008.

[ED89] P. Van Eijk and Michel Diaz, editors. *Formal Description Technique LOTOS: Results of the Esprit Sedos Project*. Elsevier Science Inc., New York, NY, USA, 1989.

[EGHK99] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: scalable coordination in sensor networks. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 263–270, New York, NY, USA, 1999. ACM.

[EIM+09] Soledad Escolar, Florin Isaila, Alejandro Mateos, Luis Sanchez garcía, and David Singh. SENFIS: a sensor node file system for increasing the scalability and reliability of Wireless Sensor Networks applications. *The Journal of Supercomputing*, 2009.

[EJL+03] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[fC03] Request for Comments 3561. Ad hoc on-demand distance vector (AODV) routing. *http://www.ietf.org/rfc/rfc3561.txt*, 2003.

[fC07a] Request for Comments 4728. The dynamic source routing protocol (DSR) for mobile ad-hoc networks for IPv4. *http://www.ietf.org/rfc/rfc4728.txt*, 2007.

[fC07b] Request for Comments 4919. 6LoWPAN: Overview, assumptions, problem statement and goals. *http://tools.ietf.org/html/rfc4919*, 2007.

[fC07c] Request for Comments 4944. Transmission of IPv6 packets over IEEE 802.15.4 networks. *http://tools.ietf.org/html/rfc4944*, 2007.

[Gal06] Sebastià Galmés. Lifetime Issues in Wireless Sensor Networks for Vineyard Monitoring. In *3rd IEEE International Conference on Mobile Ad Hoc and Sensor Systems (IEEE MASS 2006), Poster Session*, pages 542–545, Vancouver, Canada, October 2006.

[Gay] David Gay. The Matchbox File System. http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/matchbox-design.pdf.

[GBK+05] Aniruddha Gokhale, Krishnakumar Balasubramanian, Arvind S. Krishna, Jaiganesh Balasubramanian, George Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *URL: http://www.cs.wustl.edu/~schmidt/PDF/middleware-specialization.pdf*, 2005.

[GGP+03] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heidemann. An evaluation of multi-resolution storage for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 89–102, New York, NY, USA, 2003. ACM.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[Gro00] IEEE Architecture Working Group. IEEE std 1471-2000, recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000.

[Gro01] Object Management Group. OMG Unified Modeling Language specification, September 2001.

[Har07] Matus Harvan. Connecting wireless sensor networks to the Internet – a 6LoWPAN implementation for TinyOS 2.0, 2007. Adviser Jurgen Schonwalder.

[HCB00] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS*, 2000.

[Hil03] Jason Lester Hill. *System architecture for wireless sensor networks*. PhD thesis, University of California, Berkeley, 2003. Adviser-David E. Culler.

[HKKW03] V. Handziski, A. Kopke, H. Karl, and A. Wolisz. A common wireless sensor network architecture?, 2003.

[HKS+05] Chih Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[HMCP04] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18:2004, 2004.

[HPH+05] Vlado Handziski, Joseph Polastrey, Jan-Hinrich Hauer, Cory Sharpy, Adam Wolisz, and David Culler. Flexible hardware abstraction for wireless sensor networks. In *2nd European Workshop on Wireless Sensor Networks (EWSN 2005)*, Istanbul, Turkey, February 2005.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[IEE96] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language].* pub-IEEE-STD, pub-IEEE-STD:adr, 1996.

[IEE00] IEEE. The Authoritative Dictionary of IEEE Standards Term, Seventh Edition, 2000.

[IEE06] IEEE Standard for Information Technology IEEE, 802.15.4-2006. Part 15.4: Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (WPANs), 2006.

[IGE00] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67, New York, NY, USA, 2000. ACM.

[Ins08] Texas Instrument. Msp430x1xx 8mhz datasheet. http://www.ti.com/lit/gpn/msp430c1101, 2008.

[JB07] Fritz Jobst and Frank Braun. Automatic generation of syntax diagrams with a given grammar. *http://www-cgi.uni-regensburg.de/~brf09510/syntax.html*, 2007.

[JOW+02] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *In ASPLOS*, pages 96–107, 2002.

[KHB02] Joanna Kulik, Wendi Heinzelman, and Hari Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wirel. Netw.*, 8(2/3):169–185, 2002.

[KK00] Brad Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 243–254, New York, NY, USA, 2000. ACM.

[KKP99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for "smart dust". In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278, New York, NY, USA, 1999. ACM.

[KNM95] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A Flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[LC02] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM Press.

[Lev06] Phillip Levis. TinyOS Programming. *http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf*, 2006.

[LL05] Joshua Lifton and Mathew Laibowitz. Application-led research in ubiquitous computing: A wireless sensor network perspective. Munich, Germany, May 2005. Pervasive 2005. UbiApp Workshop.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[LSZM04] Ting Liu, Christopher M. Sadler, Pei Zhang, and Margaret Martonosi. Implementing software on resource-constrained mobile sensors: experiences with Impala and ZebraNet. In *MobiSys '04: Proceedings of the 2nd international conference on*

Mobile systems, applications, and services, pages 256–269, New York, NY, USA, 2004. ACM.

[Man07] D. Manjunath. A review of current operating systems for wireless sensor networks. In *Computers and Their Applications*, pages 387–394, 2007.

[Mar05] Pedro José Marrón. Middleware approaches for sensor networks. University of Stuttgart, Summer School on WSNs and Smart Objects. Schloss Dagstuhl, Aug. Germany. http://www.vs.inf.ethz.ch/events/dag2005/program/lectures/marron-2.pdf, 2005.

[MCP+02] Alan M. Mainwaring, David E. Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*, pages 88–97, 2002.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[MFjWM04] David Malan, Thaddeus Fulford-jones, Matt Welsh, and Steve Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *In International Workshop on Wearable and Implantable Body Sensor Networks*, 2004.

[MLM+05] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *In EWSN '05*, pages 278–289, 2005.

[MPar] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state-of-the-art. *ACM Computing Surveys*, To appear.

[NAD+02] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew P. Black, Peter O. Müller, Christian Zeidler, Thomas Genssler, and Reinier van den Born. A component model for field devices. In *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2002.

[NRH+09] M. Niedermayer, C. Richter, J. Hefer, S. Guttowski, and H. Reichl. SENESCOPE: A design tool for cost optimization of wireless sensor nodes. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 313–324, Washington, DC, USA, 2009. IEEE Computer Society.

[PHC04] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for Wireless Sensor Networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM Press.

[Pin04] Jim Pinto. Intelligent sensor networks. *Automation World*, 2004.

[Raj05] Ioannis Daskalopoulos Hamadoun Diall Kishore Raja. Power-efficient and reliable MAC for Routing in Wireless Sensor Networks, 2005.

[Rey93] John C. Reynolds. The discoveries of continuations. *Lisp Symb. Comput.*, 6(3-4):233–248, 1993.

[RKM02] Kay Romer, Oliver Kasten, and Friedemann Mattern. Middleware challenges for Wireless Sensor Networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):59–61, 2002.

[RL03] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless Sensor Networks and applications*, pages 60–67, New York, NY, USA, 2003. ACM.

[RM04] Kay Römer and Friedemann Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004.

[ROGLA06] Venkatesh Rajendran, Katia Obraczka, and J. J. Garcia-Luna-Aceves. Energy-efficient, collision-free medium access control for wireless sensor networks. *Wirel. Netw.*, 12(1):63–78, 2006.

[RWA+08] Injong Rhee, Ajit Warrier, Mahesh Aia, Jeongki Min, and Mihail L. Sichitiu. Z-MAC: a hybrid MAC for wireless sensor networks. *IEEE/ACM Trans. Netw.*, 16(3):511–524, 2008.

[Sad07] Daniel A. Sadilek. Prototyping domain-specific languages for wireless sensor networks. ATEM 07: 4th International Workshop on Software Language Engineering, 2007.

[SG08] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):1–29, 2008.

[SGJ08] Yanjun Sun, Omer Gurewitz, and David B. Johnson. RI-MAC: a receiver-initiated asynchronous duty cycle MAC protocol for dynamic traffic loads in wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 1–14, New York, NY, USA, 2008. ACM.

[SHrC+04] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 188–200, New York, NY, USA, 2004. ACM.

[Tat05] Simon Tatham. Simon Tatham's implementation of coroutines in C. http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html, 2005.

[TDHV09] Nicolas Tsiftes, Adam Dunkels, Zhitao He, and Thiemo Voigt. Enabling large-scale storage in sensor networks with the Coffee File System. In *Proceedings of The 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, page 12, San Francisco, California, USA, 2009.

[Tec] Crossbow Technology. MTS/MDA Sensor and Data Acquisition Board User's Manual. http://www.intel.com/research/exploratory/motes.htm.

[Tit05] Ben L. Titzer. Avrora: Scalable sensor network simulation with precise timing. In *In Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN*, pages 477–482, 2005.

[uSW05] uSWN. uSWN: Solving Major Problems in MicroSensorial Wireless Networks. https://www.uswn.eu/j/, 2005.

[VFE+06] Thiemo Voigt, Niclas Finne, Joakim Eriksson, Adam Dunkels, and Fredrik Österlind. Cross-level sensor network simulation with COOJA. In *Proceedings of First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, page 8, Tampa, Florida, USA, 2006.

[VP02] Ledeczi A. Volgyesi P. Component-based development of networked embedded applications. In *28th Euromicro Conference, Component-Based Software Engineering Track*, Dortmund, Germany, 2002.

[WALJ+06] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 381–396, Berkeley, CA, USA, 2006. USENIX Association.

[WB97] Mark Weiser and John Seely Brown. The coming age of calm technology. pages 75–85, 1997.

[YGE01] Yan Yu, Ramesh Govindan, and Deborah Estrin. Geographical and energy aware routing: a recursive data dissemination protocol for Wireless Sensor Networks. Technical report, Technical Report UCLA/CSD-TR-01-0023, UCLA Computer Science Dept., May 2001.

[YHE02] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings 21st International Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, USA, 2002.